Automatic Detection of Hazards
in
Digital Logic Circuits

by

Neil C. Erdwien

BSEE, Kansas State University, 1984

A THESIS

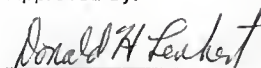submitted in partial fulfillment of the
the requirements for the degree

MASTER OF SCIENCE

Electrical Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1989

Approved by:

Donald H. Lenhert

Major Professor

ABSTRACT

This thesis describes SYMSIM, a symbolic simulator for digital logic circuits that is capable of detecting all static and dynamic hazards in combinational circuits. SYMSIM is based on a conventional simulator with a realistic propagation delay model that provides distinct propagation delays for different gate types as well as separate rise and fall times. An exhaustive search algorithm drives the simulator to examine all hazard possibilities. Time values are maintained as symbolic values during simulation, hence the name SYMSIM. Hazards are propagated through the network, allowing hazard detections to be suppressed if the hazard never reaches an output. These methods could be extended to sequential circuits although the execution time would suffer.

ACKNOWLEDGMENTS

I must first acknowledge Dr. Don Lenhert for his guiding hand in getting this project done and his patience in getting it written down.

Secondly, I would like to thank the management of KSU's Computing and Tele-communications Activities for providing an environment that encourages academic excellence.

Finally and most importantly, I am in debt to my wife Tiffany, whose sacrifices for this project are greater than my own.

CONTENTS

- v -

FIGURES

TABLES

INTRODUCTION

Since the early days of digital logic circuitry, computers have been used to analyze and evaluate new designs. This thesis describes a computer program capable of analyzing a digital logic circuit for a class of design errors.

The program, named SYMSIM for Symbolic Simulator, is capable of detecting *all* static and dynamic hazards in a combinational circuit. If this algorithm finds no faults, the designer is guaranteed that no errors of this class exist in the circuit. Alternative methods of detecting hazards require test vectors and do not guarantee complete coverage.

The ability to mechanically detect hazards will both speed up the design process and result in more robust designs. As in any situation where automatic techniques displace manual methods, the designer's attention can be focused on other aspects, leading to more creative and complex designs.

This thesis consists of a series of increasingly detailed descriptions of the algorithms used by SYMSIM. This section explains only the most basic capabilities. "Background" on page 7 adds additional information by comparing and contrasting to alternative methods. "SYMSIM Overview" on page 12 is a short overview of SYMSIM's methods. "The SYMSIM Circuit Model" on page 18 and "Exhaustive

- 1 -

Search Strategy" on page 32 describe in detail the two unique algorithms used by SYMSIM. "SYMSIM Programmer's Guide" on page 57 describes some internal features of SYMSIM while "SYMSIM User's Guide" on page 62 describes SYMSIM from a user's viewpoint. Finally, the ultimate description is the SYMSIM source code which is listed in "Source Code for SYMSIM" on page 69.

1.1 Definitions

A *hazard* is a transient glitch created when a circuit's steady-state behavior differs from its transient behavior[4,6]. SYMSIM only deals with hazards caused by either a single signal or the combination of two signals.

Hazards can be classified into *static* and *dynamic* hazards. A signal that *should* remain static, but in fact has a period of uncertainty, has a static hazard. Static hazards are caused by the combination of two signals which always have complementary steady-state values but may have the same value during a transition period [6].

A signal that *should* undergo a single transition, but in fact may "bounce" during the transition has a dynamic hazard. Dynamic hazards are caused by the combination of a static hazard and two signals which always have the same steady-state values but may have the same complementary values during a transition period [6].

The static/dynamic classification is further divided by the ending state of the signal. The resulting taxonomy divides hazards into four types, static 0, static 1, dynamic 0, and dynamic 1. This division is depicted in Figure 1.

- 2 -

Figure 1: Hazard taxonomy.
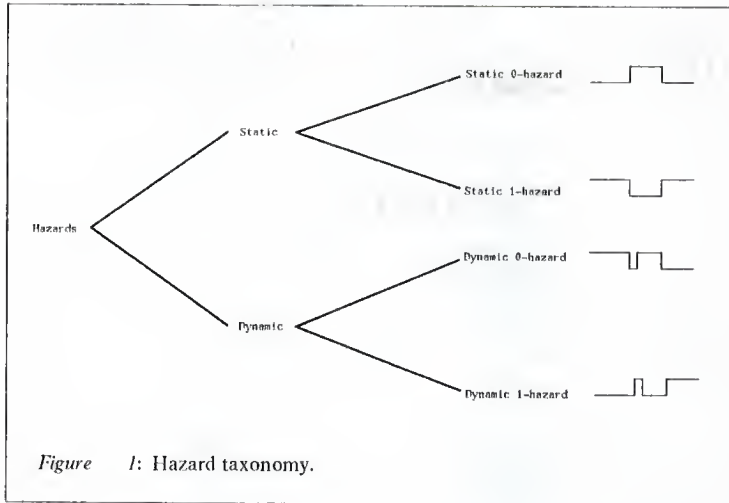
The term *circuit element* is used synonymously with *gate*. Both refer to a primitive, atomic element of the circuit, i.e., the smallest unit of simulation.

Transitions and hazards are known by several names. Typically, transitions are said to be either 0-1 or 1-0 transitions. However, because neither "0-1" nor "1-0" are valid identifiers in conventional programming languages, the terms 'up' and 'down' are used for program input and output. The terms *rising* and *falling* are also used to refer to 0-1 and 1-0 transitions.

The terms *leading* and *trailing* are used to refer to the first and last edge of a transition, regardless of whether the transition is rising or falling.

- 3 -

Also, SYMSIM uses `st0`, `st1`, `dy0`, and `dy1` as shorthand for "static 0-hazard" through "dynamic 1-hazard", respectively.

Conventional notation is used to refer to propagation delay values, i.e., T_{PLH} refers to the time needed for a Low-to-High or rising transition, while T_{PHL} corresponds to a High-to-Low or falling transition.

1.2 Hazard Examples

Consider the circuit shown in Figure 2, which is the simplest circuit with a static 0-hazard.
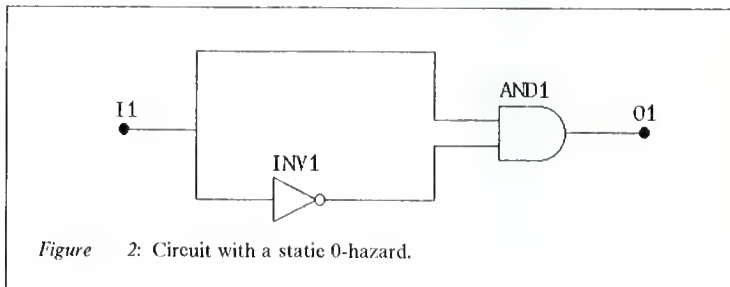
Figure   2: Circuit with a static 0-hazard.

The output of this circuit will always be 0 under steady-state conditions. In other words, because the input signal and its complement are fed to the AND gate, one of the two will always be 0, and hence the output of the gate will always be 0.

The situation is different in a transient analysis. Assume that the input has been at a 0 level long enough for transients to die out, then switches to a 1. The AND gate

will momentarily produce a 1 output because the top input line will switch to a 1 before the bottom input line switches to a 0. The cause of the timing difference is the propagation delay of the inverter. The result is a static 0-hazard.

The root cause of this hazard is that the minimum delay for a 0-1 transition along the top path is less than the maximum delay for a 1-0 transition along the bottom path. Including the possible delay caused by the transition time of the input signal, the hazard exists because

$$T_{PLHmin}(\text{top}) < T_{PHLmax}(\text{bottom}) + T_{PLHmax}(\text{II})$$

For the opposite transition, a 1-0 transition, no hazard is present in the circuit shown. Because the top line is assumed to have zero propagation delay, the 0 value will always reach the AND gate before the inverter can produce a 1. However, if the maximum propagation delay for a 1-0 transition on the top line is greater than the minimum delay along the bottom path, a hazard results for a 1-0 transition also. In other words, a hazard exists if

$$T_{PHLmax}(\text{top}) + T_{PHLmax}(\text{II}) > T_{PLHmin}(\text{bottom})$$

It is possible to meet both conditions and have a hazard for both transitions. Changing the top line in Figure 2 to contain two inverters and assuming propagation delay values for the 74ALS logic family results in both hazards.

Figure 3 shows a circuit with a dynamic 1-hazard.

A similar analysis shows the problem in this circuit. When the input is stable at 0, INV1 will produce a 1, NAND1 will produce a 1, and AND1 will stabilize at 0. When the input switches to a 1, the output of AND1 will switch to a 1 after only

- 5 -
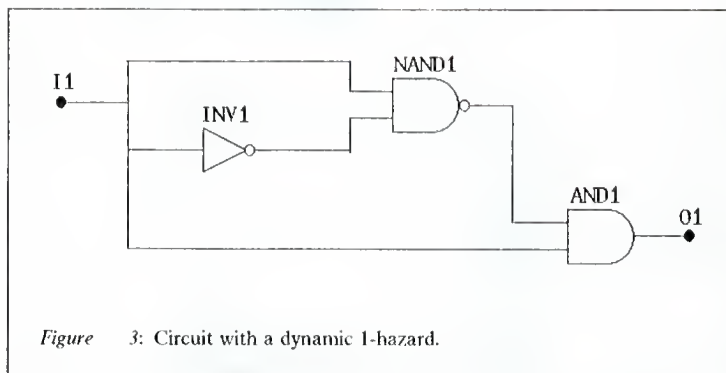
Figure 3: Circuit with a dynamic 1-hazard.

the propagation delay of AND1 itself. INV1 and NAND1 combine to create a static 1-hazard, meaning that the output of NAND1 will momentarily drop to 0 before stabilizing at 1. This drop will cause AND1 to drop also, then return to 1. The net result is the sequence 0, 1, 0, 1 on the output of AND1, which is a dynamic 1-hazard.

Chapter II

BACKGROUND

2.1   Hazard Detection through Simulation

The conventional approach to finding hazards is to use a simulator. A human prepares an input sequence, i.e., a sequence of K N-tuples to provide input to an N-input circuit. These values are used to drive a simulator, which determines the circuit's output values versus time. Typically the designer would have to check the output sequences for anomalies.

This approach has two major disadvantages. First, a human must determine the input sequence to be used. In practice, this means that the human must have a knowledge of where hazards exist in the circuit and what input sequence will trigger the problem.

Second, like using sample input data to test software, this approach can never prove the absence of hazards -- just the existence of hazards.

SYMSIM can do better. The circuit description itself contains all the information required to analytically determine the presence or absence of hazards. A human, given enough patience and accuracy, can fully analyze a circuit. Digital computers, known for their patience and accuracy, should be able to fully analyze a circuit also.

To do this analysis, a computer program must be able to "think" about a digital logic circuit at a higher level than a simulator.

Nonetheless, the basis for SYMSIM is a detailed simulator with a realistic timing model. One can view SYMSIM as a simulator with a director that automatically generates the input test vectors that exercise the entire circuit.

2.2 Circuit Models

The earliest and simplest simulators used two-level logic, i.e., every signal was either a 0 or a 1. Often there was an implicit third state called "unknown". However in the early simulators this wasn't a state in its own right, it was simply the absence of any known state.

Eichelberger[1] and Yoeli and Rinon[2] developed true three-valued systems, although this concept dates back to Muller[3]. In such a system the "unknown" state is a state in its own right, i.e., it propagates through circuit elements just like 0's and 1's. This system is capable of detecting static hazards under the worst-case assumption that any gate may have any propagation delay from 0 to infinity.

Fantauzzi[5] included both static and dynamic hazards as true states, thus generating the 9-level system shown in the left column of Table 1. The truth tables for every circuit element must be elaborated to a 9 by 9 matrix. This system is capable of detecting both static and dynamic hazards under the same assumptions about propagation delay as in Eichelberger's system.

Table 1: Correspondence between Fantauzzi's 9 levels and SYMSIM's 5 levels.

9-level state	Equivalent		
0	0		
1	1		
0-1	0	0-1	1
1-0	1	1-0	0
unk	unk		
static 0	0	unk	0
static 1	1	unk	1
dynamic 0	1	unk	0
dynamic 1	0	unk	1

Both these systems rely on the user to create a sequence of input values used to exercise the circuit.

SYMSIM uses a combination of the 9-level system and a restricted 5-level system where each signal is represented by a sequence of three or more values. Both the first and the last must be either unknown, 0, or 1. The middle state(s) can be either unknown, 0, 1, 0-1, or 1-0.

The semantics carried in the 9-level can be duplicated using this 5-level sequence. Table 1 shows the correspondence between SYMSIM's 5-level system and Fantauzzi's 9-level system.

The advantage of this 5-level sequence rather than a simpler 9-level system is that the 5-level sequence allows time information to be added. Two time values are kept for each sequence -- the earliest time that the first transition can occur and the latest time that the second transition can occur. This is discussed in detail in "The SYMSIM Circuit Model" on page 18.

2.3   Algebraic Detection of Hazards

There are algebraic tests that can be used to detect hazards with no simulation[6].
Like the simulation methods mentioned earlier, these methods assume that any cir-
cuit element may have any time delay from 0 to infinity. They also must take a
worst-case view of network connectivity. In other words, if there is a propagation
path from point A to point B, the analytic methods will assume that path is ener-
gized, even if intermediate gates block the path in crucial circumstances. Figure 4
shows an example of a circuit that has a hazard according to algebraic methods, but
in fact has no hazard.

Figure   4: Example of algebraic method limitation.
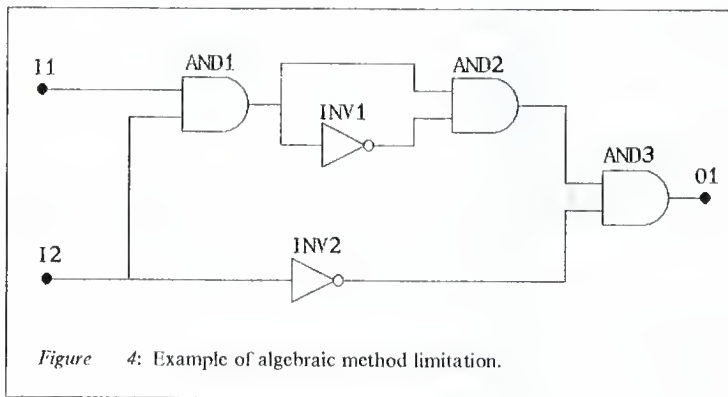
The problems associated with current algebraic methods lead McCluskey to con-
clude that "Algebraic methods are of very little value in analyzing production net-
works."[6]

2.4 Related Tools

The concepts used in SYMSIM are closely related to those used in test vector generation and timing verification. Test vector generation is the creation of a suite of input values that are used to verify the correct fabrication of a circuit. SYMSIM tests for *design* errors, while test vectors check for *manufacturing* errors. However, both have the characteristic of being limited to using input values to test for errors within the circuit.

Timing verifiers perform an analysis of the timing of a circuit in order to locate slow paths. Two varieties of timing verifiers are in common use -- dynamic and static. Dynamic verifiers are essentially simulators with their attendant need for test vectors. Static verifiers are analogous to the methods described in "Algebraic Detection of Hazards" on page 10 in that no test vectors are needed but false positive indications occur. Hybrid verifiers are now appearing that combine characteristics of both.[9]

- 11 -

Chapter III

SYMSIM OVERVIEW

This chapter presents an overview of the process SYMSIM uses to detect hazards. Two techniques used by SYMSIM are distinctive and require explanation -- the circuit model used by the simulator and the search strategy used to check for hazards.

The simulator is the foundation of the process and the search strategy generates the simulator inputs needed to make a complete exploration of potential trouble spots.

To test a preliminary design with SYMSIM, a gate-level description of the circuit must be created in SYMSIM's input language, called the Trivial Circuit Format (see "The Trivial Circuit Format" on page 66). The primitive circuit elements in this description are limited to Inverters, AND, OR, NAND, NOR, and XOR gates, although each gate can have any number of inputs. Other gate types could be added, although the program is limited to N input, single output gates. Using the assumption that each input gate type is associative, the simulator evaluates each gate by combining the inputs pairwise.

The TCF file is used by SYMSIM to create an internal representation of the circuit. The simulator portion of SYMSIM can accurately model the behavior of the circuit in response to inputs. The search strategy described below drives the simulator to test every possible hazard-generating situation.

3.1 Design Goals

1. Prove the concept of the use of symbolic time information in the analysis of circuits.

2. Minimize the input required from the designer, i.e., require no test vectors.

3. Minimize the false-positive information sent to the designer. This involves reporting neither hazards that are theoretically possible but do not occur with actual time values nor hazards that do not propagate to an output of the circuit.

4. Use an accurate propagation delay model, i.e., one that provides for different delays for different gate types as well as separate rise and fall times.

3.2 Circuit Model

The simulator is based on the 9-level logic system developed by Fantauzzi[5]. Truth tables for the primitive gate types in the 9-level system are calculated by a separate program using a 5-level sequence method, which is needed to incorporate time information into the simulation. The method and the resulting tables are described in the next chapter.

The truth tables give the 9-level result of an operation on a pair of 9-level inputs. Since gates simulated by SYMSIM can have N inputs, the inputs are simulated in pairs and combined for the overall result.

Propagation delay information is included from an external file to allow the user to easily change values. The format of the file is described in "Propagation Delay File Format" on page 64.

3.3   Search Strategy

Consider the task of analyzing the circuit in Figure 5 for the presence of hazards.

Figure   5: Circuit with N inputs and M outputs.
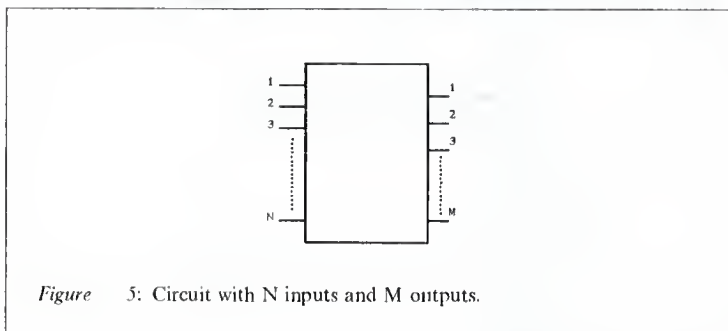
SYMSIM accomplishes this task by testing all possible transitions and all possible time alignments on relevant pairs of inputs.

Before the simulation starts, a scan is made of the circuit to determine the static connectivity. By using fan-out information, N recursive depth-first traversals of the circuit are done, starting with each input in turn. During the traversal for a given input, each gate visited is on a path from that input. In other words, the given input partially determines the gate's output. Each gate contains a bit string used to store the connectivity information. As each gate is visited, the bit corresponding to the input that started this traversal is set. The end result is that each gate has a record of which inputs can affect the gate's output. This information is used to determine which input will be dealt with next during the simulation.

- 14 -

After the preliminary traversals, the simulation for Figure 5 proceeds as follows. SYMSIM first assumes that input 1 has a 0-1 transition at time 0. The consequences of this assumption are determined. In other words, every gate whose value can be determined with just this information is simulated.

If gates remain whose output cannot be determined, a second assumption is needed. The static connectivity information and the results of the simulation so far are used to determine which input to make an assumption about. In the simulation so far, a record is kept of "dead ends" or gates whose output could not be determined. This list is used to determine which input will affect the most dead ends.

An integer counter is kept for each input and initialized to zero. Each dead end is examined, and the counter corresponding to every input that can affect the dead end is incremented. The next assumption is made about the input that is still unknown and has the highest count. Using this heuristic causes SYMSIM to rapidly make progress through the circuit.

The gate selected by this heuristic, chosen to be input 2 for this example, it is assumed to be a 0. Like the previous assumption, the consequences of this assumption are explored. A third, fourth, or more assumptions may be needed and are performed by a recursive subroutine. Eventually the process returns to the assumption that the second input is 0. This assumption is changed to a value of 1, and the process repeats.

When the second recursive exploration ends, the second input is assumed to have a 0-1 transition at a symbolic time T. Like the previous assumptions, the consequenc-

es of this are explored, possibly with the aid of additional assumptions. Finally, a 1-0 transition at time T is assumed and explored.

The entire search process is a recursive traversal of the search tree shown in Figure 6. Note that there can only be one symbolic time transition active at a time, i.e., in a path from the root to a leaf node in Figure 6.

Also, the heuristic described above may result in entire subtrees bypassed because the corresponding input didn't affect the portion of the circuit being examined.

Figure 6: The full search tree.
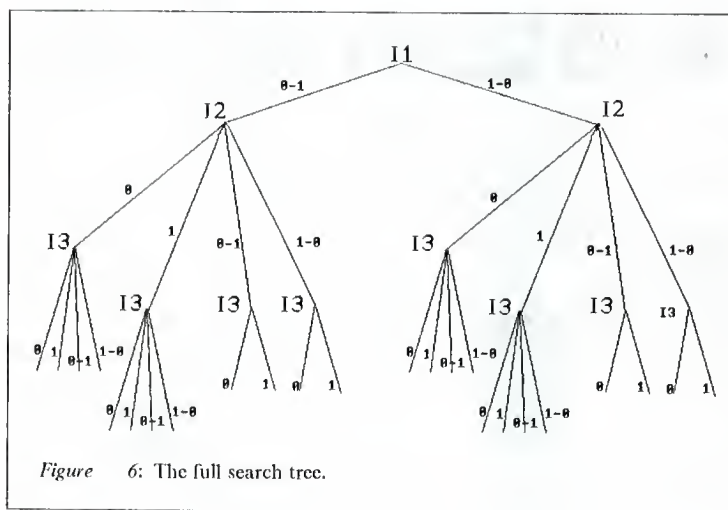
When an assumption is a transition, i.e., either a 0-1 or a 1-0 at time T, simulation will determine whether any value of T can possibly yield a hazard. A range of possible T values is kept that is narrowed down to produce specific cases.

- 16 -

A comprehensive test involves assuming both a 0-1 and a 1-0 transition for each input in turn. This first transition is always assumed to happen at time = 0 with no loss of generality. All secondary assumptions are handled in a recursive manner that guarantees complete coverage of the possibilities.

Chapter IV

THE SYMSIM CIRCUIT MODEL

This chapter describes the techniques used by SYMSIM for straight simulation. As has already been mentioned ("Circuit Models" on page 8), SYMSIM uses a hybrid of Fantauzzi's 9-level system and a sequence of three values taken from a 5-level universe. The 9-level system allows for a concise representation of values while the 5-level division into a sequence of values allows easy calculation and visualization of simulation results.

SYMSIM uses the best features of each system by representing logic values with the 9-level system. The 5-level system is used to calculate the truth tables for the 9-level system and as a means of visualizing the overall process.

Table 1 showed the mapping from a 9-level value into a 5-level sequence. The reverse mapping is not as simple, because calculation of a gate's output may result in a sequence not listed in Table 1. There will always be corresponding 9-level value, however.

The state of every node in the circuit is represented by a 9-level output value and two time variables, T_1 and T_2, that specify the earliest and latest times of uncertainty. For the three constant values, 0, 1, and unknown, the time values are irrelevant. For the other six values, the first time represents the earliest possible first transition

time and the second time represents the latest possible last transition time. For example, the value 0-1 is guaranteed to be 0 for all time prior to T_1. Likewise, after T_2 the value is 1. For $T_1 \le t \le T_2$ the value is 0-1.

The time values arise from propagation delays values that are read from a file. The time units used in this file determine the units used throughout SYMSIM. Time values are kept with one digit of precision after the decimal point. Since time values are typically measured in nanoseconds, this results in a precision of 0.1 of a nanosecond. However, any units may be used.

4.1   Example of 5-level Simulation

Consider the determination of the output of the gate shown in Figure 7(a) in the absence of any timing delays.

The output of an AND gate with a static 0-hazard and a dynamic 1-hazard as inputs is not intuitively obvious. To clarify the situation, a timing diagram similar to Figure 7(b) is created. Each 9-level value on an input is converted to the corresponding 5-level sequence and aligned according to the T_1 and T_2 values. The circuit element is evaluated independently for each vertical time slice. For each slice, the output of the gate is easy to determine because each input is always either a 0, 1, 0-1, 1-0, or unknown -- the hazard states have been eliminated by placing the same information into a sequence of states.

Once the output sequence has been determined, the pattern is analyzed and turned back into an equivalent 9-level value. In this example, the pattern 0-0-Unk-0-0 is
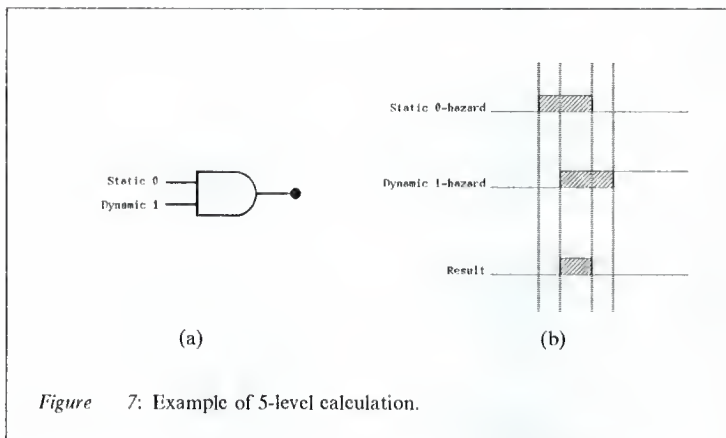
Figure 7: Example of 5-level calculation.

translated to a static 0-hazard. This is an example of a 5-level sequence that does not appear in Table 1 but nonetheless has an equivalent 9-level value.

In summary, converting 9-level values into 5-level sequences has two advantages.

1. The results of a simulation can be easily calculated and visualized, and

2. Information about the timing of the transitions and hazards can be incorporated into the simulation.

However, the result calculated with a 5-level approach can be converted back into a 9-level value.

4.2   Timing Complications

There are three complications caused by including time information that were glossed over in the above example. The input signal alignment, output signal alignment, and propagation delay complications are discussed in the following three sections.

4.2.1   Input Signal Alignment

There are six different alignments possible with two 5-level sequences as shown in Figure 8. For the purposes of illustration in the previous example, an arbitrary alignment was chosen. Chosing the proper alignment during the actual simulation is easy because the time values are known. The time alignment of the input signals is as important as the actual signal values for determining the gate's output.

When symbolic time values are involved, the range of permissible T_A values is restricted to fit the signals into each of Figure 8's six cases in turn. The simulation proceeds with the more restrictive range.

When an edge of one signal exactly coincides with an edge of the other, the signals are not considered to overlap. In other words, no zero-width pulse results from coincident edges.

Figure 8: Six possible time alignments.
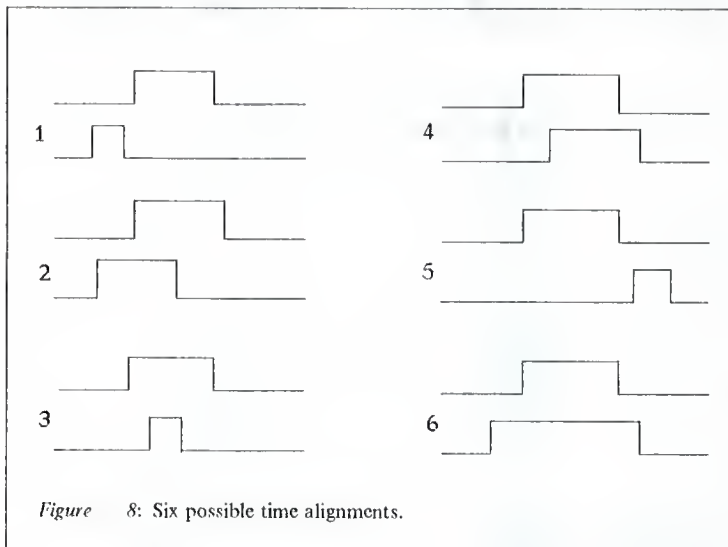
4.2.2 Output Signal Alignment

The second timing complication glossed over is that the output sequence transition times may come from either input sequence. In the example, the beginning time for the output sequence comes from the second input (the dynamic 1-hazard), while the ending time comes from the first input (the static 0-hazard). In other words, the output's T_1 comes from the second input's T_1, while the output's T_2 comes from the first input's T_2. The timing of the output sequence is as important to the simulation as the actual output value.

4.2.3 Propagation Delay

The final timing complication arises from the propagation delay of the circuit element being simulated. The real output T_1 and T_2 would not exactly match any of the input transitions. Instead, the time of the leading edge of the output signal, T_1, would be increased by the minimum rise or fall time of the gate being simulated. Similarly, T_2 would be increased by the maximum rise or fall time.

Table 2 lists which delay value is added for each of the 9 possible simulation results.

Table 2: Propagation delay time adjustment.		
9-level state	Added to T_1	Added to T_2
0	0	0
1	0	0
0-1	min rise time	max rise time
1-0	min fall time	max fall time
unknown	0	0
static 0	min rise time	max fall time
static 1	min fall time	max rise time
dynamic 0	min fall time	max fall time
dynamic 1	min rise time	max rise time

4.3 Truth Table Construction

Although the 5-level sequence is good for understanding the simulation process, the calculation is time-consuming. Therefore, SYMSIM calculates the result of simulating a gate with a table lookup using the 9-level values of the inputs. There is no reason other than speed to use the tables -- SYMSIM could use the long process.

An auxiliary program, TTABLE, runs through all combinations of input values and time alignments and creates tables of dimensions 9 by 9 by 6 for 2-input AND, OR, and XOR gates. These tables specify the output of the given gate for all combinations of the 2 inputs.

Inverters are handled by the much simpler table shown in Table 3. NAND and NOR gates are handled with the AND and OR tables followed by an inversion.

Table 3: Truth table for an Inverter.

Input	Output
zero	one
one	zero
up	down
down	up
unk	unk
st0	st1
st1	st0
dy0	dy1
dy1	dy0

The tables created by TTABLE are used to initialize the tables used by SYMSIM at compile time. In other words, TTABLE's output is C source code that is incorporated into SYMSIM. This procedure is described under "Creation of 9-Level Simulation Tables" on page 58.

The tables for AND, OR, and XOR gates are shown in Table 4 through Table 6, respectively.

- 24 -

Table 4: Truth table for an AND gate.

```
/*                         1          2          3          4          5          6        */
/*                        XXX        XXX        XXX        XXX        XXX        XXX       */
/*   X     Y      YY       YY         YY         Y          YY         YY       YYYYY      */
/*zero,zero*/ zero,0,0,  zero,0,0,  zero,0,0,  zero,0,0,  zero,0,0,  zero,0,0,
/*zero, one*/ zero,0,0,  zero,0,0,  zero,0,0,  zero,0,0,  zero,0,0,  zero,0,0,
/*zero,  up*/ zero,0,0,  zero,0,0,  zero,0,0,  zero,0,0,  zero,0,0,  zero,0,0,
/*zero,down*/ zero,0,0,  zero,0,0,  zero,0,0,  zero,0,0,  zero,0,0,  zero,0,0,
/*zero, unk*/ zero,0,0,  zero,0,0,  zero,0,0,  zero,0,0,  zero,0,0,  zero,0,0,
/*zero, st0*/ zero,0,0,  zero,0,0,  zero,0,0,  zero,0,0,  zero,0,0,  zero,0,0,
/*zero, st1*/ zero,0,0,  zero,0,0,  zero,0,0,  zero,0,0,  zero,0,0,  zero,0,0,
/*zero, dy0*/ zero,0,0,  zero,0,0,  zero,0,0,  zero,0,0,  zero,0,0,  zero,0,0,
/*zero, dy1*/ zero,0,0,  zero,0,0,  zero,0,0,  zero,0,0,  zero,0,0,  zero,0,0,
/* one,zero*/ zero,0,0,  zero,0,0,  zero,0,0,  zero,0,0,  zero,0,0,  zero,0,0,
/* one, one*/  one,0,0,   one,0,0,   one,0,0,   one,0,0,   one,0,0,   one,0,0,
/* one,  up*/   up,0,0,    up,0,0,    up,0,0,    up,0,0,    up,0,0,    up,0,0,
/* one,down*/ down,0,0,  down,0,0,  down,0,0,  down,0,0,  down,0,0,  down,0,0,
/* one, unk*/  unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,
/* one, st0*/  st0,0,0,   st0,0,0,   st0,0,0,   st0,0,0,   st0,0,0,   st0,0,0,
/* one, st1*/  st1,0,0,   st1,0,0,   st1,0,0,   st1,0,0,   st1,0,0,   st1,0,0,
/* one, dy0*/  dy0,0,0,   dy0,0,0,   dy0,0,0,   dy0,0,0,   dy0,0,0,   dy0,0,0,
/* one, dy1*/  dy1,0,0,   dy1,0,0,   dy1,0,0,   dy1,0,0,   dy1,0,0,   dy1,0,0,
/*  up,zero*/ zero,0,0,  zero,0,0,  zero,0,0,  zero,0,0,  zero,0,0,  zero,0,0,
/*  up, one*/   up,1,1,    up,1,1,    up,1,1,    up,1,1,    up,1,1,    up,1,1,
/*  up,  up*/   up,1,1,    up,1,1,    up,0,1,    up,0,0,    up,0,0,    up,0,0,
/*  up,down*/ zero,0,0,   st0,1,0,   st0,1,0,   st0,1,0,   st0,1,0,   st0,1,0,
/*  up, unk*/  unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,
/*  up, st0*/ zero,0,0,   st0,1,0,   st0,0,0,   st0,0,0,   st0,0,0,   st0,1,0,
/*  up, st1*/   up,1,1,   dy1,1,1,   dy1,1,1,   dy1,1,1,   dy1,1,1,   dy1,1,1,
/*  up, dy0*/ zero,0,0,   st0,1,0,   st0,1,0,   st0,1,0,   st0,1,0,   st0,1,0,
/*  up, dy1*/   up,1,1,   dy1,1,1,   dy1,0,1,   dy1,0,0,   dy1,0,0,   dy1,1,0,
/*down,zero*/ zero,0,0,  zero,0,0,  zero,0,0,  zero,0,0,  zero,0,0,  zero,0,0,
/*down, one*/ down,1,1,  down,1,1,  down,1,1,  down,1,1,  down,1,1,  down,1,1,
/*down,  up*/  st0,0,1,   st0,0,1,   st0,0,1,   st0,0,1,   st0,0,1,   st0,0,1,
/*down,down*/ down,0,0,  down,0,0,  down,1,0,  down,1,1,  down,1,1,  down,0,1,
/*down, unk*/  unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,
/*down, st0*/  st0,0,0,   st0,0,0,   st0,0,1,   st0,0,1,   zero,0,0,   st0,0,1,
/*down, st1*/  dy0,0,1,   dy0,0,1,   dy0,0,1,   dy0,0,1,   dy0,0,1,   dy0,0,1,
/*down, dy0*/  dy0,0,0,   dy0,0,0,   dy0,1,0,   dy0,1,1,  down,1,1,   dy0,0,1,
/*down, dy1*/  st0,0,1,   st0,0,1,   st0,0,1,   st0,0,1,   zero,0,0,   st0,0,1,
/* unk,zero*/ zero,0,0,  zero,0,0,  zero,0,0,  zero,0,0,  zero,0,0,  zero,0,0,
/* unk, one*/  unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,
/* unk,  up*/  unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,
/* unk,down*/  unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,
/* unk, unk*/  unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,
/* unk, st0*/  st0,0,0,   st0,0,0,   st0,0,0,   st0,0,0,   st0,0,0,   st0,0,0,
/* unk, st1*/  unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,
/* unk, dy0*/  unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,
/* unk, dy1*/  unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,
/* st0,zero*/ zero,0,0,  zero,0,0,  zero,0,0,  zero,0,0,  zero,0,0,  zero,0,0,
/* st0, one*/  st0,1,1,   st0,1,1,   st0,1,1,   st0,1,1,   st0,1,1,   st0,1,1,
/* st0,  up*/  st0,1,1,   st0,1,1,   st0,1,1,   st0,1,1,   zero,0,0,   st0,1,1,
/* st0,down*/ zero,0,0,   st0,1,0,   st0,1,0,   st0,1,1,   st0,1,1,   st0,1,1,
/* st0, unk*/  st0,1,1,   st0,1,1,   st0,1,1,   st0,1,1,   st0,1,1,   st0,1,1,
/* st0, st0*/ zero,0,0,   st0,1,0,   st0,1,0,   st0,0,1,   zero,0,0,   st0,1,1,
/* st0, st1*/  st0,1,1,   st0,1,1,   st0,1,1,   st0,1,1,   st0,1,1,   st0,1,1,
/* st0, dy0*/ zero,0,0,   st0,1,0,   st0,1,0,   st0,0,1,   st0,1,1,   st0,1,1,
/* st0, dy1*/  st0,1,1,   st0,1,1,   st0,0,1,   st0,0,1,   zero,0,0,   st0,1,1,
/* st1,zero*/ zero,0,0,  zero,0,0,  zero,0,0,  zero,0,0,  zero,0,0,  zero,0,0,
/* st1, one*/  st1,1,1,   st1,1,1,   st1,1,1,   st1,1,1,   st1,1,1,   st1,1,1,
/* st1,  up*/  dy1,0,1,   dy1,0,1,   dy1,0,1,   dy1,0,1,    up,0,0,   dy1,0,0,
/* st1,down*/ down,0,0,   dy0,0,0,   dy0,1,0,   dy0,1,0,   dy0,1,0,   dy0,0,0,
/* st1, unk*/  unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,
/* st1, st0*/  st0,0,0,   st0,0,0,   st0,0,0,   st0,0,0,   st0,0,0,   st0,0,0,
/* st1, st1*/  st1,1,1,   st1,1,1,   st1,1,1,   st1,1,1,   st1,1,0,   st1,1,0,
/* st1, dy0*/  dy0,0,0,   dy0,0,0,   dy0,1,0,   dy0,1,1,   dy0,1,0,   dy0,1,0,
/* st1, dy1*/  dy1,0,1,   dy1,0,1,   dy1,0,1,   dy1,0,1,   dy1,0,1,   dy1,0,1,
/* dy0,zero*/ zero,0,0,  zero,0,0,  zero,0,0,  zero,0,0,  zero,0,0,  zero,0,0,
/* dy0, one*/  dy0,1,1,   dy0,1,1,   dy0,1,1,   dy0,1,1,   dy0,1,1,   dy0,1,1,
/* dy0,  up*/  st0,0,1,   st0,0,1,   st0,0,1,   st0,0,1,   zero,0,0,   st0,0,1,
/* dy0,down*/ down,0,0,   dy0,0,0,   dy0,1,0,   dy0,1,1,   dy0,1,1,   dy0,0,1,
/* dy0, unk*/  unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,
/* dy0, st0*/  st0,0,0,   st0,0,0,   st0,0,0,   st0,0,0,   st0,0,0,   st0,0,0,
/* dy0, st1*/  dy0,0,1,   dy0,0,1,   dy0,0,1,   dy0,0,1,   zero,0,0,   dy0,0,1,
/* dy0, dy0*/  dy0,0,0,   dy0,0,0,   dy0,1,0,   dy0,1,1,   dy0,1,1,   dy0,0,1,
/* dy0, dy1*/  st0,0,1,   st0,0,1,   st0,0,1,   st0,0,1,   zero,0,0,   st0,0,1,
/* dy1,zero*/ zero,0,0,  zero,0,0,  zero,0,0,  zero,0,0,  zero,0,0,  zero,0,0,
/* dy1, one*/  dy1,1,1,   dy1,1,1,   dy1,1,1,   dy1,1,1,   dy1,1,1,   dy1,1,1,
/* dy1,  up*/  dy1,1,1,   dy1,1,1,   dy1,0,1,   dy1,0,1,    up,0,0,   dy1,1,0,
/* dy1,down*/ zero,0,0,   st0,1,0,   st0,1,0,   st0,1,0,   st0,1,0,   st0,1,0,
/* dy1, unk*/  unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,
```

- 25 -

```
/* dyl, st0*/ zero,0,0,   st0,1,0,   st0,0,0,   st0,0,0,   st0,0,0,   st0,1,0,
/* dyl, st1*/  dyl,1,1,   dyl,1,1,   dyl,1,1,   dyl,1,0,   dyl,1,0,   dyl,1,0,
/* dyl, dy0*/ zero,0,0,   st0,1,0,   st0,1,0,   st0,1,0,   st0,1,0,   st0,1,0,
/* dyl, dyl*/  dyl,1,1,   dyl,1,1,   dyl,0,1,   dyl,0,0,   dyl,0,0,   dyl,1,0
```

Table 1: Truth table for an OR gate.

```
/*                     1          2          3          4          5          6       */
/*                    XXX        XXX        XXX        XXX        XXX        XXX       */
/*   X     Y     YY          YY          Y          YY          YY          YYYYY  */
/*zero,zero*/  zero,0,0,  zero,0,0,  zero,0,0,  zero,0,0,  zero,0,0,  zero,0,0,
/*zero, one*/  one,0,0,   one,0,0,   one,0,0,   one,0,0,   one,0,0,   one,0,0,
/*zero,  up*/  up,0,0,    up,0,0,    up,0,0,    up,0,0,    up,0,0,    up,0,0,
/*zero,down*/  down,0,0,  down,0,0,  down,0,0,  down,0,0,  down,0,0,  down,0,0,
/*zero, unk*/  unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,
/*zero, st0*/  st0,0,0,   st0,0,0,   st0,0,0,   st0,0,0,   st0,0,0,   st0,0,0,
/*zero, st1*/  st1,0,0,   st1,0,0,   st1,0,0,   st1,0,0,   st1,0,0,   st1,0,0,
/*zero, dy0*/  dy0,0,0,   dy0,0,0,   dy0,0,0,   dy0,0,0,   dy0,0,0,   dy0,0,0,
/*zero, dy1*/  dy1,0,0,   dy1,0,0,   dy1,0,0,   dy1,0,0,   dy1,0,0,   dy1,0,0,
/* one,zero*/  one,0,0,   one,0,0,   one,0,0,   one,0,0,   one,0,0,   one,0,0,
/* one, one*/  one,0,0,   one,0,0,   one,0,0,   one,0,0,   one,0,0,   one,0,0,
/* one,  up*/  one,0,0,   one,0,0,   one,0,0,   one,0,0,   one,0,0,   one,0,0,
/* one,down*/  one,0,0,   one,0,0,   one,0,0,   one,0,0,   one,0,0,   one,0,0,
/* one, unk*/  one,0,0,   one,0,0,   one,0,0,   one,0,0,   one,0,0,   one,0,0,
/* one, st0*/  one,0,0,   one,0,0,   one,0,0,   one,0,0,   one,0,0,   one,0,0,
/* one, st1*/  one,0,0,   one,0,0,   one,0,0,   one,0,0,   one,0,0,   one,0,0,
/* one, dy0*/  one,0,0,   one,0,0,   one,0,0,   one,0,0,   one,0,0,   one,0,0,
/* one, dy1*/  one,0,0,   one,0,0,   one,0,0,   one,0,0,   one,0,0,   one,0,0,
/*  up,zero*/  up,1,1,    up,1,1,    up,1,1,    up,1,1,    up,1,1,    up,1,1,
/*  up, one*/  one,0,0,   one,0,0,   one,0,0,   one,0,0,   one,0,0,   one,0,0,
/*  up,  up*/  up,0,0,    up,0,0,    up,1,1,    up,1,1,    up,1,1,    up,1,1,
/*  up,down*/  st1,0,1,   st1,0,1,   st1,0,1,   st1,0,1,   one,0,0,   st1,0,1,
/*  up, unk*/  unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,
/*  up, st0*/  dy1,0,1,   dy1,0,1,   dy1,1,1,   dy1,1,1,   up,1,1,    dy1,0,1,
/*  up, st1*/  st1,0,0,   st1,0,0,   st1,0,1,   st1,0,1,   one,0,0,   st1,0,1,
/*  up, dy0*/  st1,0,1,   st1,0,1,   st1,0,1,   st1,0,1,   one,0,0,   st1,0,1,
/*  up, dy1*/  dy1,0,0,   dy1,0,0,   dy1,1,0,   dy1,1,1,   up,1,1,    dy1,0,1,
/*down,zero*/  down,1,1,  down,1,1,  down,1,1,  down,1,1,  down,1,1,  down,1,1,
/*down, one*/  one,0,0,   one,0,0,   one,0,0,   one,0,0,   one,0,0,   one,0,0,
/*down,  up*/  one,0,0,   one,0,0,   st1,1,0,   st1,1,0,   st1,1,0,   st1,1,0,
/*down,down*/  down,1,1,  down,1,1,  down,0,1,  down,0,0,  down,1,0,  down,1,0,
/*down, unk*/  unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,
/*down, st0*/  down,1,1,  dy0,1,1,   dy0,1,1,   dy0,1,0,   dy0,1,0,   dy0,1,0,
/*down, st1*/  one,0,0,   st1,1,1,   st1,1,0,   st1,0,0,   st1,1,0,   st1,1,0,
/*down, dy0*/  down,1,1,  dy0,1,1,   dy0,0,1,   dy0,0,0,   dy0,1,0,   dy0,1,0,
/*down, dy1*/  one,0,0,   dy0,1,0,   st1,1,0,   st1,1,0,   st1,1,0,   st1,1,0,
/* unk,zero*/  unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,
/* unk, one*/  one,0,0,   one,0,0,   one,0,0,   one,0,0,   one,0,0,   one,0,0,
/* unk,  up*/  unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,
/* unk,down*/  unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,
/* unk, unk*/  unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,
/* unk, st0*/  unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,
/* unk, st1*/  st1,0,0,   st1,0,0,   st1,0,0,   st1,0,0,   st1,0,0,   st1,0,0,
/* unk, dy0*/  unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,
/* unk, dy1*/  unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,
/* st0,zero*/  st0,1,1,   st0,1,1,   st0,1,1,   st0,1,1,   st0,1,1,   st0,1,1,
/* st0, one*/  one,0,0,   one,0,0,   one,0,0,   one,0,0,   one,0,0,   one,0,0,
/* st0,  up*/  up,0,0,    dy1,0,0,   dy1,1,0,   dy1,1,0,   dy1,0,0,   dy1,0,0,
/* st0,down*/  dy0,0,1,   dy0,0,1,   dy0,0,1,   dy0,0,0,   down,0,0,  dy0,0,0,
/* st0, unk*/  unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,
/* st0, st0*/  st0,0,1,   st0,0,1,   st0,1,1,   st0,1,1,   st0,1,0,   st0,0,0,
/* st0, st1*/  st1,0,0,   st1,0,0,   st1,0,0,   st1,0,0,   st1,0,0,   st1,0,0,
/* st0, dy0*/  dy0,0,1,   dy0,0,1,   dy0,0,1,   dy0,0,0,   dy0,0,0,   dy0,0,0,
/* st0, dy1*/  dy1,0,0,   dy1,0,0,   dy1,1,0,   dy1,1,0,   dy1,1,0,   dy1,0,0,
/* st1,zero*/  st1,1,1,   st1,1,1,   st1,1,1,   st1,1,1,   st1,1,1,   st1,1,1,
/* st1, one*/  one,0,0,   one,0,0,   one,0,0,   one,0,0,   one,0,0,   one,0,0,
/* st1,  up*/  one,0,0,   st1,1,0,   st1,1,0,   st1,1,1,   st1,1,1,   st1,1,1,
/* st1,down*/  st1,1,1,   st1,1,1,   st1,0,1,   st1,0,1,   one,0,0,   st1,1,1,
/* st1, unk*/  st1,1,1,   st1,1,1,   st1,1,1,   st1,1,1,   st1,1,1,   st1,1,1,
/* st1, st0*/  st1,1,1,   st1,1,1,   st1,1,1,   st1,1,1,   st1,1,1,   st1,1,1,
/* st1, st1*/  one,0,0,   st1,1,0,   st1,0,0,   st1,0,0,   one,0,0,   st1,1,0,
/* st1, dy0*/  st1,1,1,   st1,1,1,   st1,0,1,   st1,0,1,   one,0,0,   st1,1,1,
/* st1, dy1*/  one,0,0,   st1,1,0,   st1,1,0,   st1,1,1,   st1,1,1,   st1,1,1,
/* dy0,zero*/  dy0,1,1,   dy0,1,1,   dy0,1,1,   dy0,1,1,   dy0,1,1,   dy0,1,1,
/* dy0, one*/  one,0,0,   one,0,0,   one,0,0,   one,0,0,   one,0,0,   one,0,0,
/* dy0,  up*/  up,0,0,    st1,1,0,   st1,1,0,   st1,1,0,   st1,1,0,   st1,1,0,
/* dy0,down*/  dy0,1,1,   dy0,1,1,   dy0,0,1,   dy0,0,0,   down,0,0,  dy0,1,0,
/* dy0, unk*/  unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,
/* dy0, st0*/  dy0,1,1,   dy0,1,1,   dy0,1,1,   dy0,1,0,   dy0,1,0,   dy0,1,0,
/* dy0, st1*/  one,0,0,   st1,1,0,   st1,1,0,   st1,0,0,   st1,1,0,   st1,1,0,
/* dy0, dy0*/  dy0,1,1,   dy0,1,1,   dy0,0,1,   dy0,0,0,   dy0,1,0,   dy0,1,0,
/* dy0, dy1*/  one,0,0,   st1,1,0,   st1,1,0,   st1,1,0,   st1,1,0,   st1,1,0,
/* dy1,zero*/  dy1,1,1,   dy1,1,1,   dy1,1,1,   dy1,1,1,   dy1,1,1,   dy1,1,1,
/* dy1, one*/  one,0,0,   one,0,0,   one,0,0,   one,0,0,   one,0,0,   one,0,0,
/* dy1,  up*/  up,0,0,    dy0,0,0,   dy1,1,0,   dy1,1,1,   dy1,1,1,   dy1,0,1,
/* dy1,down*/  st1,0,1,   st1,0,1,   st1,0,1,   st1,0,1,   one,0,0,   st1,0,1,
/* dy1, unk*/  unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,
```

```
/* dy1, st0*/  dy1,0,1,  dy1,0,1,  dy1,1,1,  dy1,1,1,  dy1,1,1,  dy1,0,1,
/* dy1, st1*/  st1,0,0,  st1,0,0,  st1,0,0,  st1,0,1,  one,0,0,  st1,0,1,
/* dy1, dy0*/  st1,0,1,  st1,0,1,  st1,0,1,  st1,0,1,  one,0,0,  st1,0,1,
/* dy1, dy1*/  dy1,0,0,  dy1,0,0,  dy1,1,0,  dy1,1,1,  dy1,1,1,  dy1,0,1
```

Table 6: Truth table for an XOR gate.

```
/*                    1        2        3        4        5        6     */
/*                   XXX      XXX      XXX      XXX      XXX      XXX     */
/* X      Y     YY        YY        Y        YY        YY      YYYYY  */
/*zero,zero*/ zero,0,0,  zero,0,0,  zero,0,0,  zero,0,0,  zero,0,0,  zero,0,0,
/*zero, one*/  one,0,0,   one,0,0,   one,0,0,   one,0,0,   one,0,0,   one,0,0,
/*zero,  up*/   up,0,0,    up,0,0,    up,0,0,    up,0,0,    up,0,0,    up,0,0,
/*zero,down*/ down,0,0,  down,0,0,  down,0,0,  down,0,0,  down,0,0,  down,0,0,
/*zero, unk*/  unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,
/*zero, st0*/  st0,0,0,   st0,0,0,   st0,0,0,   st0,0,0,   st0,0,0,   st0,0,0,
/*zero, st1*/  st1,0,0,   st1,0,0,   st1,0,0,   st1,0,0,   st1,0,0,   st1,0,0,
/*zero, dy0*/  dy0,0,0,   dy0,0,0,   dy0,0,0,   dy0,0,0,   dy0,0,0,   dy0,0,0,
/*zero, dy1*/  dy1,0,0,   dy1,0,0,   dy1,0,0,   dy1,0,0,   dy1,0,0,   dy1,0,0,
/* one,zero*/  one,0,0,   one,0,0,   one,0,0,   one,0,0,   one,0,0,   one,0,0,
/* one, one*/ zero,0,0,  zero,0,0,  zero,0,0,  zero,0,0,  zero,0,0,  zero,0,0,
/* one,  up*/ down,0,0,  down,0,0,  down,0,0,  down,0,0,  down,0,0,  down,0,0,
/* one,down*/   up,0,0,    up,0,0,    up,0,0,    up,0,0,    up,0,0,    up,0,0,
/* one, unk*/  unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,
/* one, st0*/  st1,0,0,   st1,0,0,   st1,0,0,   st1,0,0,   st1,0,0,   st1,0,0,
/* one, st1*/  st0,0,0,   st0,0,0,   st0,0,0,   st0,0,0,   st0,0,0,   st0,0,0,
/* one, dy0*/  dy1,0,0,   dy1,0,0,   dy1,0,0,   dy1,0,0,   dy1,0,0,   dy1,0,0,
/* one, dy1*/  dy0,0,0,   dy0,0,0,   dy0,0,0,   dy0,0,0,   dy0,0,0,   dy0,0,0,
/*  up,zero*/   up,1,1,    up,1,1,    up,1,1,    up,1,1,    up,1,1,    up,1,1,
/*  up, one*/ down,1,1,  down,1,1,  down,1,1,  down,1,1,  down,1,1,  down,1,1,
/*  up,  up*/  st0,0,1,   st0,0,1,   st0,1,1,   st0,1,0,   st0,1,0,   st0,0,0,
/*  up,down*/  st1,0,1,   st1,0,1,   st1,1,1,   st1,1,0,   st1,1,0,   st1,0,0,
/*  up, unk*/  unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,
/*  up, st0*/  dy1,0,1,   dy1,0,1,   dy1,1,1,   dy1,1,0,   dy1,1,0,   dy1,0,0,
/*  up, st1*/  dy0,0,1,   dy0,0,1,   dy0,1,1,   dy0,1,0,   dy0,1,0,   dy0,0,0,
/*  up, dy0*/  st1,0,1,   st1,0,1,   st1,1,1,   st1,1,0,   st1,1,0,   st1,0,0,
/*  up, dy1*/  st0,0,1,   st0,0,1,   st0,1,1,   st0,1,0,   st0,1,0,   st0,0,0,
/*down,zero*/ down,1,1,  down,1,1,  down,1,1,  down,1,1,  down,1,1,  down,1,1,
/*down, one*/   up,1,1,    up,1,1,    up,1,1,    up,1,1,    up,1,1,    up,1,1,
/*down,  up*/  st1,0,1,   st1,0,1,   st1,1,1,   st1,1,0,   st1,1,0,   st1,0,0,
/*down,down*/  st0,0,1,   st0,0,1,   st0,1,1,   st0,1,0,   st0,1,0,   st0,0,0,
/*down, unk*/  unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,
/*down, st0*/  dy0,0,1,   dy0,0,1,   dy0,1,1,   dy0,1,0,   dy0,1,0,   dy0,0,0,
/*down, st1*/  dy1,0,1,   dy1,0,1,   dy1,1,1,   dy1,1,0,   dy1,1,0,   dy1,0,0,
/*down, dy0*/  st0,0,1,   st0,0,1,   st0,1,1,   st0,1,0,   st0,1,0,   st0,0,0,
/*down, dy1*/  st1,0,1,   st1,0,1,   st1,1,1,   st1,1,0,   st1,1,0,   st1,0,0,
/* unk,zero*/  unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,
/* unk, one*/  unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,
/* unk,  up*/  unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,
/* unk,down*/  unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,
/* unk, unk*/  unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,
/* unk, st0*/  unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,
/* unk, st1*/  unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,
/* unk, dy0*/  unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,
/* unk, dy1*/  unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,
/* st0,zero*/  st0,0,1,   st0,0,1,   st0,1,1,   st0,1,0,   st0,1,0,   st0,0,0,
/* st0, one*/  st1,1,1,   st1,1,1,   st1,1,1,   st1,1,1,   st1,1,1,   st1,1,1,
/* st0,  up*/  dy1,0,1,   dy1,0,1,   dy1,1,1,   dy1,1,0,   dy1,1,0,   dy1,0,0,
/* st0,down*/  dy0,0,1,   dy0,0,1,   dy0,1,1,   dy0,1,0,   dy0,1,0,   dy0,0,0,
/* st0, unk*/  unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,
/* st0, st0*/  st0,0,1,   st0,0,1,   st0,1,1,   st0,1,0,   st0,1,0,   st0,0,0,
/* st0, st1*/  st1,0,1,   st1,0,1,   st1,1,1,   st1,1,0,   st1,1,0,   st1,0,0,
/* st0, dy0*/  dy0,0,1,   dy0,0,1,   dy0,1,1,   dy0,1,0,   dy0,1,0,   dy0,0,0,
/* st0, dy1*/  dy1,0,1,   dy1,0,1,   dy1,1,1,   dy1,1,0,   dy1,1,0,   dy1,0,0,
/* st1,zero*/  st1,1,1,   st1,1,1,   st1,1,1,   st1,1,1,   st1,1,1,   st1,1,1,
/* st1, one*/  st0,1,1,   st0,1,1,   st0,1,1,   st0,1,1,   st0,1,1,   st0,1,1,
/* st1,  up*/  dy0,0,1,   dy0,0,1,   dy0,1,1,   dy0,1,0,   dy0,1,0,   dy0,0,0,
/* st1,down*/  dy1,0,1,   dy1,0,1,   dy1,1,1,   dy1,1,0,   dy1,1,0,   dy1,0,0,
/* st1, unk*/  unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,
/* st1, st0*/  st1,0,1,   st1,0,1,   st1,1,1,   st1,1,0,   st1,1,0,   st1,0,0,
/* st1, st1*/  st0,0,1,   st0,0,1,   st0,1,1,   st0,1,0,   st0,1,0,   st0,0,0,
/* st1, dy0*/  dy1,0,1,   dy1,0,1,   dy1,1,1,   dy1,1,0,   dy1,1,0,   dy1,0,0,
/* st1, dy1*/  dy0,0,1,   dy0,0,1,   dy0,1,1,   dy0,1,0,   dy0,1,0,   dy0,0,0,
/* dy0,zero*/  dy0,1,1,   dy0,1,1,   dy0,1,1,   dy0,1,1,   dy0,1,1,   dy0,1,1,
/* dy0, one*/  dy1,1,1,   dy1,1,1,   dy1,1,1,   dy1,1,1,   dy1,1,1,   dy1,1,1,
/* dy0,  up*/  st1,0,1,   st1,0,1,   st1,1,1,   st1,1,0,   st1,1,0,   st1,0,0,
/* dy0,down*/  st0,0,1,   st0,0,1,   st0,1,1,   st0,1,0,   st0,1,0,   st0,0,0,
/* dy0, unk*/  unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,
/* dy0, st0*/  dy1,0,1,   dy1,0,1,   dy1,1,1,   dy1,1,0,   dy1,1,0,   dy1,0,0,
/* dy0, st1*/  dy0,0,1,   dy0,0,1,   dy0,1,1,   dy0,1,0,   dy0,1,0,   dy0,0,0,
/* dy0, dy0*/  st0,0,1,   st0,0,1,   st0,1,1,   st0,1,0,   st0,1,0,   st0,0,0,
/* dy0, dy1*/  st1,0,1,   st1,0,1,   st1,1,1,   st1,1,0,   st1,1,0,   st1,0,0,
/* dy1,zero*/  dy1,1,1,   dy1,1,1,   dy1,1,1,   dy1,1,1,   dy1,1,1,   dy1,1,1,
/* dy1, one*/  dy0,1,1,   dy0,1,1,   dy0,1,1,   dy0,1,1,   dy0,1,1,   dy0,1,1,
/* dy1,  up*/  st0,0,1,   st0,0,1,   st0,1,1,   st0,1,0,   st0,1,0,   st0,0,0,
/* dy1,down*/  st1,0,1,   st1,0,1,   st1,1,1,   st1,1,0,   st1,1,0,   st1,0,0,
/* dy1, unk*/  unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,   unk,0,0,
```

- 29 -

```
/* dy1, st0*/   dy1,0,1,   dy1,0,1,   dy1,1,1,   dy1,1,0,   dy1,1,0,   dy1,0,0,
/* dy1, st1*/   dy0,0,1,   dy0,0,1,   dy0,1,1,   dy0,1,0,   dy0,1,0,   dy0,0,0,
/* dy1, dy0*/   st1,0,1,   st1,0,1,   st1,1,1,   st1,1,0,   st1,1,0,   st1,0,0,
/* dy1, dy1*/   st0,0,1,   st0,0,1,   st0,1,1,   st0,1,0,   st0,1,0,   st0,0,0
```

Each table is indexed by:

1.   The 9-level value on input 1, which is the first value listed on each line of the tables.

2.   The 9-level value on input 2, which is the second value listed on each line of the tables.

3.   The time alignment value, which is depicted graphically in the heading of each table. The numbers above each result column correspond to the alignment numbers listed in Figure 8.

Each element of the array consists of three variables:

1.   A 9-level result of the simulation.

2.   A logical variable that indicates whether the leading edge of the result is taken from the first or second input. A 1 indicates the resulting time derives from the first, or X, input.

3.   A logical variable that indicates whether the trailing edge of the result is taken from the first or second input. A 1 indicates the resulting time derives from the first, or X, input.

Table 4 can be used to determine the output of the example given in Figure 7. Both the example and the table apply to an AND gate. The inputs to the AND gate are a static 0-hazard and a dynamic 1-hazard; therefore the row that starts with '/* st0, dy1*/' is the proper row, and the fourth entry on that row corresponds to the time alignment assumed for the example. The entry 'st0,0,1' indicates that the result is a static 0-hazard, where the leading edge comes from the second input and the trailing edge comes from the first input. This matches the result derived earlier.

- 30 -

Fantauzzi[5] gives a table for the AND gate which matches the third result column of Table 4.

The primitive circuit elements of SYMSIM are *N*-input gates, and the tables are for 2-input gates. The assumption is made that the operations are associative, and that the result of a simulation can be calculated by treating the inputs in pairs.

Chapter V

EXHAUSTIVE SEARCH STRATEGY

5.1 Overview

Simply stated, SYMSIM detect hazards by evaluating all relevant combinations of input values. Each input in turn is assumed to have every potentially troublesome value. The consequences of this assumption are propagated through the network as far as possible. Eventually, either the entire circuit has been evaluated or additional assumptions are needed to make further progress.

In addition to the steady-state values of 0 and 1, SYMSIM tries combinations of both rising and falling transitions on the inputs. However, hazards supplied on the circuit's inputs are not considered because such hazards are considered problems in the driving circuit.

Secondary assumptions are made with an *arbitrary starting time value*. In other words, the time alignment of the second assumption is not fixed with respect to the first signal. Instead, a variable, T_A, is added to both numeric time values, T_1 and T_2. The numeric value of T_A is not determined until the signal interacts with the first assumption. The value of T_A is then restricted to narrow down the simulation to possible hazard-causing situations.

This arbitrary time value is the essence of SYMSIM -- the "symbolic" of "symbolic simulator" derives from it.

There are a two important limitations of this method.

First, the execution time of this algorithm may be large, depending on how interconnected the circuit is. This is discussed later under "Timing Analysis" on page 52.

Second, only one arbitrary time value is allowed. In other words, one "fixed" time and one "floating" time are simulated, but two arbitrary and independent "floating" values are not. It is perfectly acceptable for the arbitrary time signal to exist in several parts of the circuit due to fan-out. However, all the signals depend on the same value of T_A.

The consequence of this restriction is that the algorithm will only detect hazards caused by a coordinated change on two inputs. If a pattern of changes on three inputs triggers a hazard, SYMSIM will not detect it. However, such hazards are considered to be sufficiently obscure to be of little interest.

5.2   Examples

As an introduction to SYMSIM's search strategy, this section presents a series of three examples that are increasingly complex. The steps SYMSIM takes to discover the hazards in each circuit are described.

- 33 -

5.2.1   Example with No Secondary Assumptions

Consider the simple circuit shown in Figure 2, which has a static 0-hazard when a falling edge is used as input. This circuit is relatively easy to analyze because it has only one input.

SYMSIM will detect this potential hazard by trying both potential problem inputs, i.e., both a 0-1 and a 1-0 transition. There are four nodes in the circuit, I1, INV1, AND1, and O1; all are initially set to unknown.

The value 0-1 at $T_1 = 0$ and $T_2 = 1$ is assumed for I1. The 1 used as the value of T_2 is an arbitrary choice of a rise time for the input. The first gate in I1's fan-out list, AND1, is simulated. Unfortunately, it is not possible to determine AND1's output, because the output of an AND gate with inputs of 0-1 and unknown is unknown. If the output of AND1 had been determined, the simulation would proceed forward in the circuit to AND1's fan-out list. Because of this block, nothing further can be done on this branch.

The other element attached to I1, INV1, is considered next. Fortunately, the output of INV1 can be calculated -- an inverter with a 0-1 transition for input has a 1-0 transition for output.

The propagation delay of INV1 must be included also. The input 0-1 transition has $T_1 = 0$ and $T_2 = 1$. According to Table 2, the output 1-0 transition must have the minimum and maximum fall times added to T_1 and T_2, respectively. Assuming propagation delay values for the 74ALS logic family, the time values are modified to $T_1 = 0 + T_{PHLmin} = 0 + 2 = 2$ and $T_2 = 1 + T_{PHLmax} = 1 + 8 = 9$.

The simulation then proceeds forward to INV1's fan-out, which leads back to AND1. This time AND1 has a 0-1 and a 1-0 transition as inputs. The time values are such that the first input's uncertainty zone is completely before the second input's, which corresponds to the fifth column of the truth table for AND gates, Table 4.

The table entry on the row labeled `/* up,down*/` in the fifth result column is `st0,1,0`. The output of AND1 is therefore a static 0-hazard, where the output T_1 comes from the first input and T_2 comes from the second input. Remembering to add in AND1's propagation delay, the example's times are $T_1 = 0 + T_{PLHmin} = 0 + 4 = 4$ and $T_2 = 9 + T_{PHLmax} = 9 + 10 = 19$.

Finally, O1, the only gate in AND1's fan-out, is simulated. The result of simulating an output node is always exactly the node's input, hence the circuit has a static 0-hazard for $4 \le T \le 19$.

The relevant portion of SYMSIM's output when run on the circuit in Figure 2 is shown in Table 7. SYMSIM's steps can be traced by comparing to the above explanation.

Table 7 and those that follow were generated by running SYMSIM with a msglevel value of 8, which generates more verbose output than normal. See "Command Format and Options" on page 62 for information on msglevel values.

Logic values output by SYMSIM consist of two parts.
1.   A name for the appropriate 9-level state, e.g., `zero`, `one`, `up`, `down`, `st0`, etc., and
2.   The T_1 and T_2 values in square brackets.

Table 7: SYMSIM output for the circuit in Figure 2.

```
Assumption 1: lnput gate ll is set to up [0.0,1.0]
   simulating AND gate AND1
   ...result: unk
   simulating Inverter gate INV1
   ...result: down [2.0,9.0]
   simulating AND gate AND1
*** Error: st0 [4.0,19.0] on the output of gate AND1
    Caused by:
    -- assumption 1: ll set to up [0.0,1.0]
   ...result: st0 [4.0,19.0]
   simulating Output gate O1
*** Error: st0 [4.0,19.0] on output line O1
    Caused by:
    -- assumption 1: ll set to up [0.0,1.0]
   ...result: st0 [4.0,19.0]
Done with assumption 1

Assumption 1: lnput gate ll is set to down [0.0,1.0]
   simulating AND gate AND1
   ...result: unk
   simulating lnverter gate lNV1
   ...result: up [3.0,12.0]
   simulating AND gate AND1
   ...result: zero
   simulating Output gate O1
   ...result: zero
Done with assumption 1
```

After finishing with the first assumption, SYMSIM proceeds to test the circuit with the assumption that II is set to 1-0. Going through the above process determines that the output of AND1 is 0 for all time, so there is no hazard indication.

5.2.2   Example with Secondary Assumptions

Consider the circuit in Figure 9, which consists of the same circuit as in Figure 2 with the output "blocked" by an AND gate.
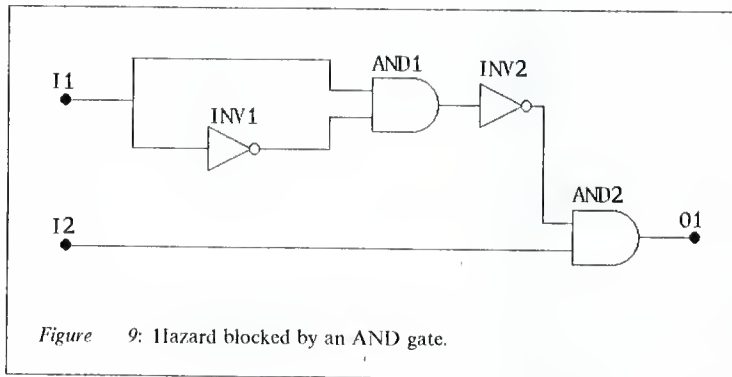
Figure   9: Hazard blocked by an AND gate.

Analysis of this circuit starts the same way as the previous circuit. In other words, the assumption that I1 is 0-1 yields a static 0-hazard on the output of AND1. The second inverter, INV2, turns this into a static 1-hazard, which is an input to AND2. However, simulation of AND2 yields an unknown. The first assumption has been taken as far as possible; another assumption is needed to make further progress. (Interestingly, INV2 is needed because the output of an AND gate with a static 0-hazard and an unknown as inputs is a static 0-hazard. However, a static 1-hazard and an unknown yield an unknown.)

The next assumption is made about the second input, I2. There are four possible assumed values for I2: 0, 1, 0-1, and 1-0.

The first assumption, 0, causes AND2 to always yield a 0, hence is quickly eliminated.

The second assumption, that I2 is a 1, allows the static 1-hazard through AND2 to the output O1. The result is a detected static 1-hazard on the output line for $9 \le T \le 44$. Table 8 shows SYMSIM's output for this analysis.

Table 8: SYMSIM output for the circuit in Figure 9.

```
Assumption 1: Input gate I1 is set to up [0.0,1.0]
    simulating ANO gate AND1
    ...result: unk
    simulating Inverter gate INV1
    ...result: down [2.0,9.0]
    simulating AND gate AND1
*** Error: st0 [4.0,19.0] on the output of gate AND1
    Caused by:
    -- assumption 1: I1 set to up [0.0,1.0]
    ...result: st0 [4.0,19.0]
    simulating Inverter gate INV2
*** Error: st1 [6.0,30.0] on the output of gate INV2
    Caused by:
    -- assumption 1: I1 set to up [0.0,1.0]
    ...result: st1 [6.0,30.0]
    simulating AND gate AND2
    ...result: unk
Assumption 2: Input gate I2 is set to zero
    simulating AND gate AND2
    ...result: zero
    simulating Output gate O1
    ...result: zero
Done with assumption 2

Assumption 2: Input gate I2 is set to one
    simulating AND gate AND2
*** Error: st1 [9.0,44.0] on the output of gate AND2
    Caused by:
    -- assumption 1: I1 set to up [0.0,1.0]
    -- assumption 2: I2 set to one
    ...result: st1 [9.0,44.0]
    simulating Output gate O1
*** Error: st1 [9.0,44.0] on output line O1
    Caused by:
    -- assumption 1: I1 set to up [0.0,1.0]
    -- assumption 2: I2 set to one
    ...result: st1 [9.0,44.0]
Done with assumption 2
    .      .      .      .      .
    .      .      .      .      .
    .      .      .      .      .
```

5.2.3   Example with Symbolic Time Information

This final example incorporates symbolic time information. Although the previous example, Figure 9, could be used to illustrate the same points, the circuit in Figure 10 illustrates the same points in more general terms.
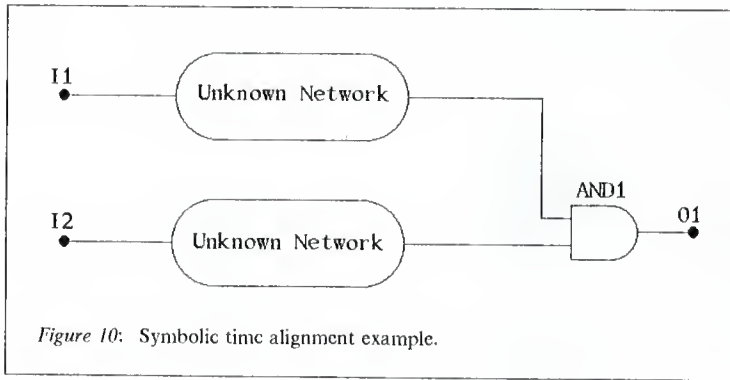
Figure 10: Symbolic time alignment example.

In the top portion of Figure 10, I1, possibly in combination with other inputs, has created a 0-1 transition for $55 \le T \le 72$. As part of the analysis, I2 is assumed to have a 0-1 transition at time T_A. This value has propagated through the network at the bottom of Figure 10 and becomes 1-0 transition for $T_A + 23 \le T \le T_A + 33$.

These two values are combined in the AND gate, however T_A has no value currently. Based on the T_1 and T_2 values of the two signals, one or more of the six time alignments listed in Figure 8 are chosen. Next, minimum and maximum values for T_A are chosen such that the output of the AND gate has a hazard, if possible.

In this example, the width of the first pulse is $72 - 55 = 17$ and the second is $33 - 23 = 10$. Because the width of the first input is larger than the second, any of the first five time alignments are possible. Referring to the '/* up,down*/' line of Table 4, the first time alignment has a result of 0, while alignments 2-5 have a result of 'st0,1,0'.

For each possible time alignment, a restriction is made on T_A and the simulation proceeds. In this case, the first time alignment has the first input's leading edge after the second input's trailing edge, which gives a time restriction of $T_A + 33 \leq 55$ or $T_A \leq 22$. With this restriction, the output of the AND gate is 0, and no hazard is generated.

The next time alignment has the second input's transition times straddling the first input's leading edge. This boils down to $T_A + 23 \leq 55$ or $T_A \leq 32$ (this keeps the second input's leading edge before the first input's leading edge). Also, $T_A + 33 \geq 55$ or $T_A \geq 22$ (this keeps the second input's trailing edge after the first input's leading edge). The net result is the restriction $22 \leq T_A \leq 32$ With this restriction, SYMSIM proceeds to generate the static 0-hazard on the output and moves forward to the output node.

The third time alignment results in the conditions $T_A + 23 \geq 55$ and $T_A + 33 \leq 72$. The overall result is that $32 \leq T_A \leq 39$.

Because the result in both cases is the same, i.e., a static 0-hazard, these restrictions can be collapsed into the single condition $22 \leq T_A \leq 39$. The other time alignments are handled similarly and can also be collapsed into the single condition $22 \leq T_A$.

- 41 -

5.3 General Description

Describing SYMSIM's search strategy in specific cases, as was done in the previous examples, never shows the complete picture. However, the examples should help provide a frame work for the current section, which describes the procedure in general terms.

First, the data structures used in SYMSIM are described. The following four sections describe four functions or subroutines within SYMSIM that interact and are important to understanding the search procedure.

5.3.1 Data Structures

When SYMSIM reads the Trivial Circuit Format file that describes the circuit to test, a network of control blocks referred to as the circuit web is built. Each node of the circuit web represents a single gate or circuit element. The result is a set of intertwined trees as shown in Figure 11. Each input to the circuit is the root of a tree that grows from left to right. The nodes on the right edge are the circuit outputs. The intermediate nodes are the Inverter, AND, OR, NAND, NOR, and XOR gates that comprise the circuit. The links in the web represent the connections between gates.

The trees rooted at each input are not disjoint. A gate that is part of the trees corresponding to two inputs is affected by both inputs.

- 42 -

Figure 11: The circuit web.
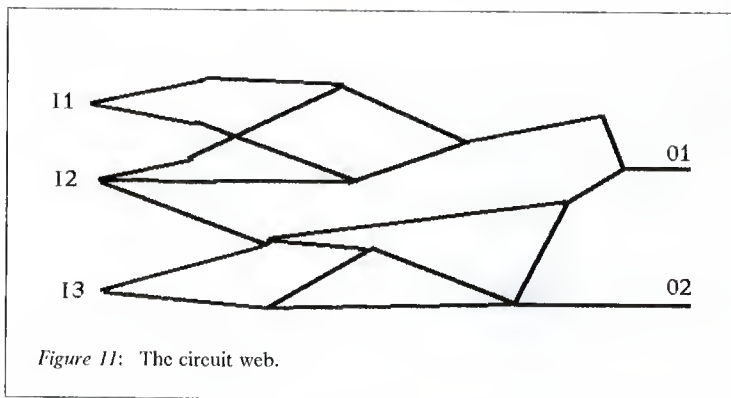
Figure 12 shows the data structures needed for a single circuit element. When the term "pointer to a gate" is used, the pointer actually points to the center box in Figure 12.

Figure 12: An element of the circuit web.
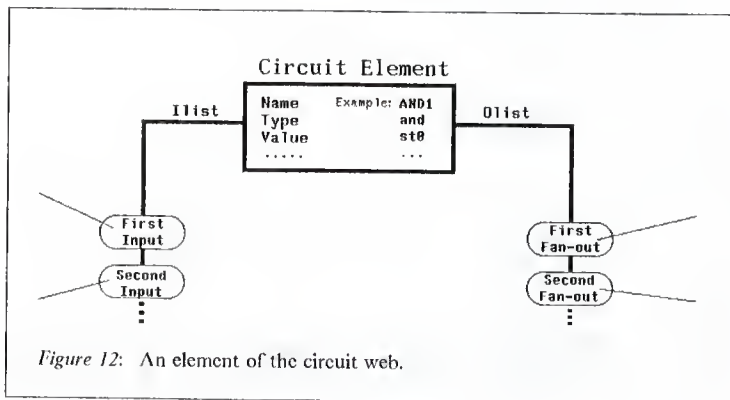
Each node has two types of links associated with it -- an input list and an output list. The input list is used to find the inputs of a gate from a pointer to the gate. Each element of the input list contains a pointer to one of the inputs of the gate.

The output list records where the gate's output goes. Each element on the output list points to a gate on this gate's fan-out.

Note that the input list has an entry for each of N inputs, but the output list represents a single output with M gates attached to it.

Each linked list contains a variable number if items -- a NULL pointer is used to mark the end of the list.

All of the search procedures follow the output list to traverse the circuit. The input list could be used to go backwards through the circuit, but this is done only to find a gate's inputs for the purpose of evaluating that gate. In other words, no traversal is done using the input list.

Because input and output elements are "circuit elements" just like AND and OR gates, they each have a data structure like Figure 12. For an input node, the input list has no elements; for an output node, the output list has no elements.

5.3.2   SIM1

The SIM1 function is the simplest of the three -- it simulates a single gate. SIM1 is passed a pointer to the gate to be simulated. The gate's inputs are retrieved, and the 9-level tables like Table 4 are consulted to determine the gate's output.

SIM1 never calls itself or the other two procedures listed here.

- 44 -

5.3.3　FORWARD

Like the SIM1 function, FORWARD is passed a pointer to a single gate. FORWARD calls the SIM1 function to evaluate the indicated gate, *and calls itself to evaluate the gate's children*.

In other words, FORWARD performs a recursive depth-first traversal of the circuit web. FORWARD's traversal ends when it reaches an output node (which has no children) or SIM1 fails to determine the output of a gate.

When FORWARD returns from simulating gate G, the consequences of the output of gate G have been fully explored.

Logic values are thus propagated *forward* through the circuit.

In pseudocode, FORWARD is implemented as follows.

```
Forward(gate) {

    result = sim1(gate);

    push this gate onto the stack

    if (result <> unknown) {
        for each element attached to gate's output
            Forward(attached gate);
    }

}
```

5.3.4   ASSUME

The most complex of these three functions is ASSUME, which is passed both a pointer to a gate and an assumed value. ASSUME is first called from the MAIN routine. Secondary assumptions are made when ASSUME calls itself.

ASSUME makes the assumption that the indicated gate has the indicated value, and then *fully explores the consequences of the assumption.*

After an assumption, the FORWARD procedure is called to find the other gates that can be determined as a result of the assumption.

When the single assumption is exhausted, ASSUME calls itself recursively to make secondary assumptions, if needed. The result is a mixture of two recursive depth-first traversals, one implemented by FORWARD and one by ASSUME. FORWARD traverses the circuit web shown in Figure 11, and ASSUME traverses the search tree shown in Figure 6. This structure is depicted in Figure 13.

In pseudocode, ASSUME is implemented as follows.
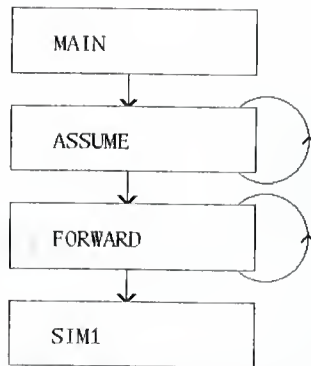
Figure 13: Recursive calling tree of SYMSIM.

```
Assume(gate, value) {

    gate->value = value;
    push this gate onto the stack

    for each element attached to gate's output
       Forward(attached gate);

    if (more remains to explore) {

       /* determine which input to assume next */
       zero the input counter array
       for each unknown dead end {
          increment the count for each input
          that affects this gate
       }
       Find the unknown input that has the
       largest count

       Assume(next input, 0);
       Assume(next input, 1);
       if (no symbolic time assumption in effect) {
          Assume(next input, 0-1);
          Assume(next input, 1-0);
       }
    }

    Pop the stack to remove the effects
    of this assumption

}
```

Both ASSUME and FORWARD must keep a record of what assumptions have been made and the resulting consequences. This is needed in order to remove the effects of an assumption. To accomplish this, a stack is used as shown in Figure 14. Both FORWARD and ASSUME push items onto this stack. To remove the consequences of an assumption, items are removed from the top of the stack until the corresponding ASSUME entry is found. The gate corresponding to each stack item is reset to unknown.

- 48 -
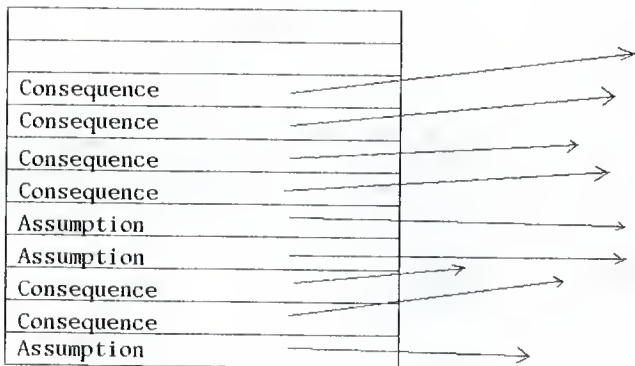
Figure 14: Stack used to record assumptions and consequences.

5.3.5   MAIN

The top-most piece of pseudocode hasn't been presented yet. This is the main program that calls ASSUME to start the recursive traversal.

In pseudocode:

```
Main() {

    process options
    read the circuit file

    for each input {
       Assume(input, 0-1);
       Assume(input, 1-0);
    }

}
```

The largest circuit that SYMSIM has been tested on is the 74181 4-bit Arithmetic Logic Unit shown in Figure 15 obtained from [12].

Figure 15: 74181 Arithmetic Logic Unit.
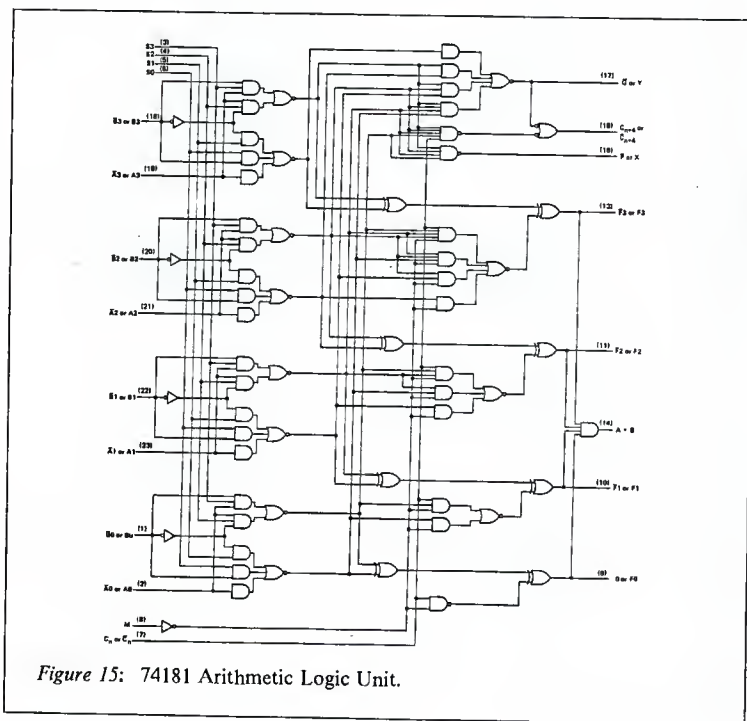
The analysis of this circuit yields a large number of trivial problems caused by joining of two input signals in a single gate. Consider the top gate of the first column of AND gates in Figure 15, which I have denoted as B3-AND1 in the following discussion. When S3 is, for example, a rising edge, B3 a 1, and A3 a falling edge, the result of B3-AND1 is a static 0-hazard. An excerpt of SYMSIM's output for this case is shown in Table 9

```
                  Table 9:  Abbreviated SYMSIM output for a 74181.

Assumption 1: Input gate S3 is set to up [0.0,1.0]
    simulating AND gate B0-and1
    ...result: unk
    simulating AND gate B1-and1
    ...result: unk
    simulating AND gate B2-and1
    ...result: unk
    simulating AND gate B3-and1
    ...result: unk
    .    .    .    .    .    .    .    .
    .    .    .    .    .    .    .    .
    .    .    .    .    .    .    .    .

Assumption 2: Input gate B3 is set to one
    .    .    .    .    .    .    .    .
    .    .    .    .    .    .    .    .
    .    .    .    .    .    .    .    .

Assumption 3: Input gate A3 is set to up tA+[0.0,1.0]
    simulating AND gate B3-and1
    ...result: st0 [4.0,11.0]
    .    .    .    .    .    .    .    .
    .    .    .    .    .    .    .    .
    .    .    .    .    .    .    .    .
```

Such an indication in this case is unavoidable and is not an indication of a real error. It is, however, an example of a false-positive indication.

6.1 Timing Analysis

SYMSIM was designed to perform a complete analysis at the expense of execution time. As such, SYMSIM's execution time can be terrible. For a pathological circuit with N inputs and P gates, the execution time may be as bad as $O(N^2 P)$

This is based on the assumption that N initial assumptions may be needed, each of which causes $N-1$ secondary assumptions, and the results propagated through P gates.

However, the realistic time should not be as bad because in a realistic circuit every gate's output does not depend on every input. As such, it is possible to bypass assumptions about a given input if that input doesn't affect any of the places that have unknown values.

6.2 Memory Analysis

SYMSIM's memory use is more tightly bounded. A control block is needed for each circuit element -- approximately 32 bytes per gate plus 8 for each input and output.

Memory is also needed for the stack of remembered assumptions and consequences, but that is also limited by the number of circuit elements. Each stack element contains a pointer to a gate.

- 52 -

Chapter VII

RECOMMENDATIONS FOR FUTURE WORK

The SYMSIM approach to simulation offers hope for more highly automated hazard detection. However, much remains to be done before SYMSIM or a program like it is a standard part of a designer's tools.

The most important and obvious extension is to allow sequential circuits. This could most easily be done by using the Huffman model to separate the combinational section of the circuit from the memory elements[11]. The output of each memory element is treated as an input to the combinational circuit. This should be a fairly straight-forward extension, although the simulation execution time will suffer because the circuit effectively has more inputs.

A firmer theoretical basis for SYMSIM would give greater confidence in SYMSIM's results. In other words, being able to *prove* that SYMSIM's algorithms will detect all hazards would be a significant addition.

Additional heuristic methods used to speed up SYMSIM would be most welcome. Using an algebraic method to determine where potential trouble spots are and then using the full SYMSIM method on only a fraction of the circuit would help considerably. Also, the procedure used to determine whether a hazard can propagate to an output could be improved by using the D-algorithm from test vector generation[4].

- 53 -

SYMSIM's symbolic time concepts could also be applied to the task of timing verification, a subject that suffers from many of the same problems as hazard detection[9].

Finally, there are several extensions that would be needed for a production system, but are not needed to prove the concept of using symbolic time information.

1. Expansion of the logic value system to include states such as the high impedance state of tri-state logic elements.

2. Provision for switch-level simulation in addition to, or as a replacement for, the current gate-level simulation.

3. Provision for a mechanism to allow the designer to limit the simulation and its output. In other words, a way for a designer to specify "Line B will not change until at least 3 nanoseconds after line A changes", and "Hazards created on line C are irrelevant."

4. Provision for input circuits in an industry-standard format. Most likely this would be implemented by a translator from the standard format into SYM-SIM's TCF.

5. Addition of the ability to use different propagation delay values for gates with different numbers of inputs. For example, a real-world 8-input AND gate has different delays than a 2-input AND gate, although SYMSIM's current circuit model treats them as identical.

Chapter VIII

SUMMARY AND CONCLUSIONS

SYMSIM has met its design goals listed in "Design Goals" on page 13, although a couple of unforseen complications arose.

The first goal was to prove the concept of symbolic time calculations during simulation. SYMSIM uses this idea to its advantage and, as a result, can detect all hazards in a circuit.

The second goal was to minimize the required input from the designer. In other words, require no test vectors, unlike conventional simulation techniques. SYMSIM doesn't require any input other than the circuit description itself.

The third goal was to minimize unwanted false-positive output. The anticipated cause of unwanted output, hazards that do not propagate to an output, are handled by SYMSIM. However, two phenomenon remain to cause unwanted output. First, most hazards *do propagate to an output*. In other words, even though SYMSIM won't report a hazard that doesn't appear on an output, almost all hazards do propagate to an output. Second, trivial hazards are possible whenever two inputs are connected to the same gate by simply aligning the input transitions to cause a hazard.

In summary, the symbolic techniques used in SYMSIM are powerful tools that can be used to provide the circuit designer with feedback about the design. Additional work is needed to make SYMSIM both general enough and fast enough for production use, although the potential benefits are large.

Appendix A

SYMSIM Programmer's Guide

This section describes some of SYMSIM's features and details that are of interest only to a programmer interested in porting SYMSIM to a new computer or extending SYMSIM's capabilities.

A.1   Operating Environments

SYMSIM is written entirely in C and was developed using the ANSI standard C compatible Microsoft C compiler on an MS-DOS machine. However, SYMSIM was designed to be portable across a wide spectrum of machines, operating systems, and C compilers. Table 10 shows the environments on which SYMSIM has been operated.

Table 10: SYMSIM Development Environments.

Machine	Operating System	C Compiler
IBM AT	MS-DOS 3.21	Microsoft 5.1
Sun 3/60	SUNOS 4.0.1	Standard CC
AT&T 3B15	AT&T System V	Standard CC

Although SYMSIM is written in ANSI standard C, two concessions were made to promote portability.

First, the variable argument list facilities provided by ANSI C are not available in all the environments listed in Table 10, but the equivalent facilities from ATC Unix System V are available. The only function that uses a variable argument list is the FATAL function that issues terminal error messages.

Second, the symbol 'ANSI' is defined to the C preprocessor and is used in a few places to generate ANSI-specific code.

A.2   Creation of 9-Level Simulation Tables

A separate program named TTABLE uses the 5-level sequence logic system to create the 9-level tables used by SYMSIM during simulation. Three tables are used -- one each for the AND, OR, and XOR gates. Each table is created by a separate invokation of TTABLE and is stored in a separate file. The commands shown in Table 11 create the necessary files, named 'AND.TBL', 'OR.TBL', and 'XOR.TBL'.

Table 11: Commands for 9-level table creation.

```
TTABLE  AND.TCF  >AND.TBL
TTABLE  OR.TCF   >OR.TBL
TTABLE  XOR.TCF  >XOR.TBL
```

These tables are included by the C language preprocessor's '#include' mechanism in the subroutine TSIM1, hence the above commands must be issued before TSIM1 is compiled.

A.3   MS-DOS Compilation

Two makefiles, one for TTABLE and one for SYMSIM, are supplied for use with the Microsoft MAKE command. The following steps are needed to generate the executable under MS-DOS.

1. Move all the files to an MS-DOS computer that has the Microsoft C compiler installed.

2. Create TTABLE with the command 'MAKE TTABLE'.

3. Issue the commands in Table 11 to generate the 9-level truth tables.

4. Create SYMSIM with the command 'MAKE SYMSIM'.

Compilers other than the Microsoft Optimizing C compiler may be suitable, however none have been tested.

A.4   Unix Compilation

A single makefile is used to create both TTABLE and SYMSIM under Unix. The following steps are needed to generate the executable under Unix.

1. Move all the files to a Unix computer. Table 10 lists the tested Unix systems, although others would be suitable.

2.	Modify the makefile to indicate either a Berkeley Unix implementation or an AT&T System V implementation. This involves defining the C preprocessor symbol 'BSD' or 'SYSV' as directed by the comments at the top of the make-file. If an ANSI compiler is being used, the 'ANSI' symbol must also be defined.

3.	Create TTABLE, the truth tables, and SYMSIM with the command 'make'.

## A.5	Expansion of Limitations

Like any computer program, SYMSIM has some arbitrary limitations that can be expanded if desired. Primary among these is the limitation on the allowed circuit elements. If the selection of Inverter, AND, OR, NAND, NOR, and XOR is limiting, it can be expanded. This would be necessary if it is desired to allow different propagation delays for devices with a different number of inputs, because then a 2-input AND gate must have a different type than a 3-input AND gate.

To make this change, the following pieces of the program must be altered.

1.	FUNC2STR must be changed to convert the new device types into strings.

2.	STR2FUNC must be changed to convert strings into the new device types.

3.	The GTYPE enumerated type must include the new devices.

4.	The MAXGTYPES manifest constant must be increased to allow for more devices.

5.	Time information files and the function that reads them must be altered.

Nothing in the simulation procedures needs to be changed.

There are other limitations that can be changed by simply altering the corresponding declaration and recompiling.

1.   The maximum number of inputs is set to 64 by the manifest constant MAXINPUTS.

2.   The maximum number of outputs is set to 64 by the manifest constant MAXOUTPUTS.

3.   The maximum stack size is set to 1024 by the manifest constant STACKSIZE.

Appendix B

SYMSIM User's Guide

B.1   Command Format and Options

SYMSIM has only two options -- and they would normally be allowed to default. The options must precede the circuit file name and must be introduced with a minus sign ('-'), even in MS-DOS where the normal option-indicating character is a slash ('/'). Blanks can optionally be used to separate the option introducer and the actual value.

The format of the SYMSIM command is:

```
SYMSIM  [-m msglevel]  [-t timefile]  file
```

where

-m   The '-m' option is followed by an integer that determines how much detail is output. The default level, 3, causes normal error messages and hazard detections that reach primary outputs to be printed. Larger numbers cause more detail to be printed while smaller numbers cause more terse output. Numbers larger than 6 are only useful when debugging SYMSIM itself.

The following chart lists 'msglevel' values and the corresponding levels of output.

0   Level 0 produces only the title, copyright, and (possibly) error messages. Even hazard detections are not reported.

3   Level 3 is the default. Hazard detections that reach primary outputs are displayed.

4   Level 4 lists hazards discovered anywhere in the circuit regardless of whether they propagate to an output.

6   Level 6 produces a listing of assumptions.

7   Level 7 dumps a description of the input file that can be used to verify the accuracy of the description of the input circuit. The propagation delay values are also echoed out.

8   Level 8 produces information about the simulation of each circuit element. Each simulation is preceded with the gate type and name and followed by the result.

9   Level 9 causes information regarding the 5-level sequence used to calculate simulation responsed to be printed. (This is effective for TTABLE only.)

-t   The '-t' option is followed by the name of the file to be used for propagation delay information. The default is 'timeinfo', a file that contains information for the 74ALS series of digital logic. The format of this file is described in "Propagation Delay File Format" on page 64.

file   The last parameter on the command line is the file containing the circuit description. The file must be in Trivial Circuit Format -- see "The Trivial Circuit Format" on page 66.

B.2   Propagation Delay File Format

SYMSIM obtains propagation delay information from an external file which can be customized by the user.

The format of the time information file is very simple. The first line of the file is a title line, which is printed when the file is read.

On the following lines are the actual time values separated by white space i.e., blanks, tabs, and carriage returns. Like the Trivial Circuit Format, comments can be included by using a pound sign ('#'), which causes the rest of the line to be ignored.

There are six circuit element types used by SYMSIM, and each has four time values associated with it -- minimum and maximum times for both falling and rising transitions. Therefore, 24 time values are required.

The four time values for each gate must be grouped together in the order T_{PLHmin}, T_{PLHmax}, T_{PHLmin}, and T_{PHLmax}. The data for the inverter must be first, followed by the data for the AND, OR, NAND, NOR, and XOR gates in order.

B.2.1   Example

Table 12 is the default propagation delay file, which contains data for the 74ALS logic family obtained from [8].

```
                Table 12:  Propagation delay file for 74ALS logic family.

74ALS TTL prop. delays, Logic Detabook, vol. II, Nat. Semi. 1984

# The first line of this file is a title line which is printed
# when the file is read.

# Characters (like these) following a pound sign (#) are ignored.

# Six sets of data follow, one for each type of gate used by
# SYMSIM.  The date must be listed in the order below, i.e.,
# Inverter, AND, OR, NAND, NOR, and finally XOR.

# TPLHmin  TPLHmax  TPHLmin  TPHLmax      Function      IC Number
    3.0     11.0     2.0      8.0     # Inverter      DM74ALS04
    4.0     14.0     3.0     10.0     # AND           DM74ALS08
    3.0     14.0     3.0     12.0     # OR            DM74ALS32
    3.0     11.0     2.0      8.0     # NAND          DM74ALS00
    3.0     12.0     3.0     10.0     # NOR           DM74ALS02
# This next one is worst-case for the two cases of whether
# the other input is high or low.
    3.0     17.0     3.0     12.0     # XOR           DM74ALS32

# The above numbers are for VCC=4.5V-5.5V, RL=500 ohms, CL=50 pF
```

B.2.2   Additional Files

Additional propagation delay information files are available. The following files are
listed in "Sample Time Information Files" on page 139.

timeinfo.als   This file is a duplicate of the default 'timeinfo' file.

timeinfo.as   This file provides information for the Advanced Schottky (AS) family.

timeinfo.ls   This file provides information for the Low-power Schottky (LS) fami-
                ly.

timeinfo.0   The 'timeinfo.0' file contains all zero propagation times -- this is
                useful for tracing SYMSIM's actions.

B.3   The Trivial Circuit Format

The Trivial Circuit Format (TCF) is the language used to describe the circuit-to-be-analyzed to SYMSIM. As its name implies, TCF is a simple format that is not suitable for generic circuit description, but is well suited to SYMSIM's needs.

A TCF file consists of a sequence of entries, one per circuit element. Each entry consists of a series of tokens delimited by white space, i.e., blanks, tabs, and carriage returns. Comments can be included in the file by using a pound sign ('#') -- this makes the rest of the line into a comment. Nothing in TCF is case-sensitive.

The tokens that make up an entry are:

1.   The keyword 'ELEMENT'.

2.   A user-specified name for the element. This name is used to specify which elements connect to which. Names can consist of any number of non-blank characters.

3.   A keyword that indicates the type of element. The valid choices are: 'Input', 'Output', 'Inverter', 'AND', 'OR', 'NAND', 'NOR', and 'XOR'.

4.   A list of zero or more names that the output of this element is connected to.

Figure 16 gives the syntax diagram for a TCF file.

This simple format has several limitations:

1.   The choice of circuit elements is limited, although the selection allows any circuit to be described.

2.   Although each gate type may have any number of inputs, all the inputs are interchangeable.

3.   Each circuit element may have only one output, although it may be tied to as many other devices as desired.

An 'Output' node cannot be used to drive other nodes; similarly, no gate's output may drive an 'Input' node.

- 66 -
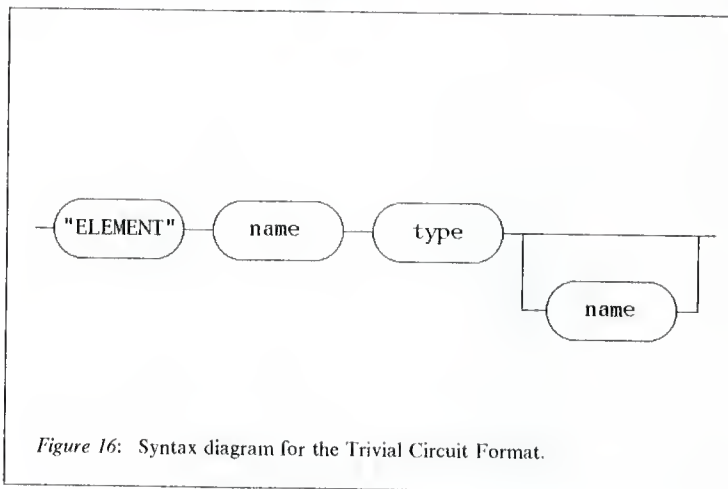
Figure 16: Syntax diagram for the Trivial Circuit Format.

B.3.1 Example

The circuit in Figure 17 shows the schematic of an XOR gate built from AND, OR, and Inverter gates. The corresponding TCF file is shown in Table 13.
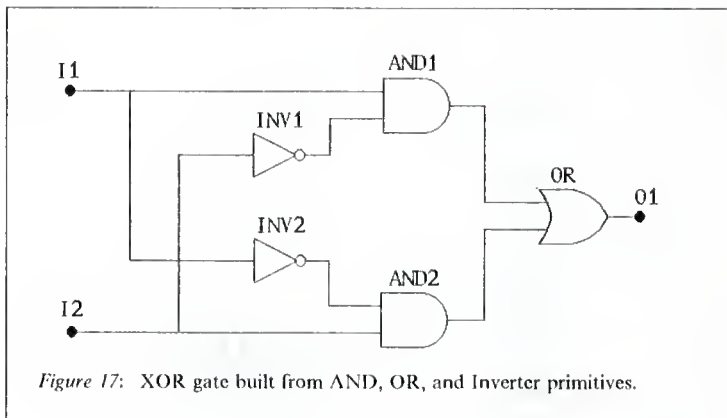
Figure 17: XOR gate built from AND, OR, and Inverter primitives.

Table 13: Trivial Circuit Format representation of Figure 17.

```
# An XDR gate built from ANDs, DRs, and INVERTERs

element I1 input
        INV2 AND1
element I2 input
        INV1 AND2
element INV1 inverter
        AND1
element INV2 inverter
        AND2
element AND1 and
        DR
element AND2 and
        DR
element OR or
        D1
element D1 output
```

Appendix C

Source Code for SYMSIM

C.1   SYMSIM.C

```
/*==================================================================*/
/*                                                                  */
/* SYMSIM -- check for hazards using symbolic simuletion            */
/*                                                                  */
/* This is the main program for SYMSIM, a symbolic simulator.       */
/* SYMSIM reads a gate-level description of a circuit in the        */
/* Trivial Circuit Format.  A complete hazard check is done by      */
/* simulating the circuit for all possible pairs of transitions     */
/* on the inputs.  Symbolic time calculations are used to test      */
/* all possible alignments of the input transitions.                */
/*                                                                  */
/*                                                                  */
/* Copyright (c) 1988, 1989                                         */
/*          Neil Erdwien and Kansas State University                */
/*          All rights reserved.                                    */
/*                                                                  */
/*==================================================================*/


/*==================================== Standard C include files */
#ifdef ANSI
#include <stddef.h>
#include <stdlib.h>
#endif
#include <stdio.h>
#include <math.h>
#include <malloc.h>
#include <string.h>


/*==================================== Program-specific include files */
#include "symsim.h"


/*==================================== Global external variables */
int      msglevel;
GBLOK    *Iptr[MAX1NPUTS];
GBLOK    *Optr[MAX1NPUTS];
char     *TimeInfoFilename;
char     *CircuitFilename;
int      Acount   = 1;
```

- 69 -

```
int       Amax;
int       VisitedValue = 0;

/*================================= External function declarations */
#ifdef ANSI
extern int getopt(int argc, char **argv, char *options);
#endif


/*==================================== Private function prototypes */
#ifdef ANSI
void ProcessArgs(int argc, char **argv);
#endif


void
ProcessArgs(argc, argv)

int    argc;
char   **argv;

{

    int         c;
    extern char *optarg;
    extern int  optind, opterr;
    int         errflg = 0;

    TimeInfoFilename = "timeinfo";
    Amax = 999;
    msglevel = 3;

    while ((c = getopt(argc, argv, "m:t:")) != -1)
        switch (c) {
        case 'm':
            msglevel = atoi(optarg);
            break;
        case 't':
            TimeInfoFilename = optarg;
            break;
        case '?':
            errflg++;
        }

    if (errflg || optind >= argc) {
        fprintf(stderr,
                "Usage: SYMSIM [-m msglevel] [-t timefile] file\n");
        exit(2);
    }

    CircuitFilename = argv[optind];

    if (optind+1 < argc)
        printf("Extra parameters ignored\n");

}
```

```
main(argc, argv)

int    argc;
char   **argv;

{

    int     i;
    int     rc;
    SEQUENCE seq;

    puts("SYMSIM version 1.0");
    puts(
        "(c) Copyright 1989 Neil Erdwien, Kansas State University\n");

    ProcessArgs(argc, argv);

    ReadTimeInfo(TimeInfoFilename);

    rc = GetCircuit(CircuitFilename, lptr, Optr);
    if (rc != 0)
        fatal("Error reading circuit '%s'", CircuitFilename);

    AddInputInfo(lptr);

    if (msglevel >= 7)
        Snap(Iptr);

    seq.t1 = 0;
    seq.t2 = 10;
    seq.tA = 0;

    for (i = 0; lptr[i] !=NULL; i++) {
        seq.val = up;
        Assuma(lptr[i], seq);

        saq.val = down;
        Assume(lptr[i], seq);
    }

    exit(0);
}
```

C.2   SYMSIM.H

```
/*====================================================================*/
/*                                                                    */
/* SYMSIM -- header file for all SYMSIM programs                       */
/*                                                                    */
/* This header file dalcares many constants, data types, global       */
/* variables, and functions used in SYMSIM.                           */
/*                                                                    */
/* This file must be included in every source program for SYMSIM.     */
/*                                                                    */
/*                                                                    */
```

- 71 -

```
/* Copyright (c) 1988, 1989                                          */
/*          Neil Erdwien and Kansas State University                 */
/*          All rights reserved.                                     */
/*                                                                   */
/*==================================================================*/


/*================================ Manifest constant definitions */
#define MAXINPUTS 64
#define MAXOUTPUTS 64


/*================================== Data Structure Declarations */
/* TIME is the type associated with all time variables           */
#define TIME long
#define INITIAL -1
#define MAXTIME 99999999

/* The VALUE type holds a 9-level logic value.                   */
/* Several routines depend on the order of these values.         */
typedef enum {
    zero,
    one,
    up,
    down,
    unk,
    st0,
    st1,
    dy0,
    dy1,
} VALUE;

/* A 5-level sequence is held in the SEQUENCE type.              */
typedef struct {
    VALUE val;
    TIME  t1;
    TIME  t2;
    int   tA;
} SEQUENCE;

/* The basic circuit elements are determined by the GTYPE type.  */
#define MAXGTYPES 9
typedef enum {
    input,
    output,
    inverter,
    and,
    or,
    nand,
    nor,
    xor,
    invalid
} GTYPE;

/* The type BITSTRING is used for the bit string holding static  */
/* connectivity information.                                     */
typedef unsigned short BITSTRING[MAXINPUTS/16];
```

- 72 -

```c
/* This is the main data structure used to represent each gate.   */

typedef struct gblok {

   GTYPE          function;   /* Type of this circuit element         */
   char           *neme;      /* Name of this circuit element         */
   SEQUENCE       seq;        /* Current value of this output         */
   int            visited;    /* Visited flag for global traversels   */
   BITSTRING      causes;     /* Bit mask of input effects            */

struct glue {
   struct glue *next;
   struct gblok *gate;
}
   *ilist,
   *olist;

} GBLOK;


/* The linked lists used to record input and output information   */
/* are of type GLUE.                                              */

typedef struct glue GLUE;


/*===================================== Global external variables */
extern int      Acount;
extern int      Amax;
extern int      VisitedValue;
extern GBLOK    *lptr[MAXINPUTS];
extern int      msglevel;
extern int      HaveTAassumption;
extern TIME     TAmin, TAmax;

/* The stack is used to record assumptions and the consequences   */
/* resulting.                                                     */
#define STACKSIZE 1024
extern  GBLOK    *Stack[STACKSIZE];
extern  int      StackPtr;
#define push(gate) { Stack[StackPtr++] = gete; \
                    essert(StackPtr < STACKSIZE); }
#define pop         ( StackPtr > 0 ? Stack[--StackPtr] : NULL )


/*================================= External function declarations */

#ifdef ANSI

int      GetCircuit   (cher *filename, GBLOK *Iptr[],GBLOK *Optr[]);
char     *GetToken     (FILE *fp);
void     Snap         (GBLOK *Iptr[]);
void     ReadTimeInfo (char *);
SEQUENCE Val2Seq      (VALUE value);
VALUE    Seq2Val      (SEQUENCE sequence);
char     *Val2Str      (VALUE value);
char     *Seq2Str      (SEQUENCE sequence);
char     *Func2Str     (GTYPE);
```

```
char      *Time2Str    (TIME);
GTYPE      Str2Func    (char *string);
void       AddInputInfo (GBLOK *Iptr[]);
GBLOK     *AllocGblok  (GTYPE function, char *name);
void       Siml        (GBLOK *gate);
void       Forward     (GBLOK *gate);
void       Assume      (GBLOK *gate, SEQUENCE seq);
void       AssmPrt     (void);
void       fatal       (char *, ...);

VALUE      stubINP     (VALUE inputs[], int numinputs);
VALUE      stubINV     (VALUE inputs[], int numinputs);
VALUE      stubAND     (VALUE inputs[], int numinputs);
VALUE      stubOR      (VALUE inputs[], int numinputs);
VALUE      stubNAND    (VALUE inputs[], int numinputs);
VALUE      stubNOR     (VALUE inputs[], int numinputs);
VALUE      stubXOR     (VALUE inputs[], int numinputs);


#else


int        GetCircuit();
char      *GetToken();
void       Snap();
void       ReadTimeInfo();
SEQUENCE   Val2Seq();
VALUE      Seq2Val();
char      *Val2Str();
char      *Seq2Str();
char      *Func2Str();
char      *Time2Str();
GTYPE      Str2Func();
void       AddInputInfo();
GBLOK     *AllocGblok();
void       Siml();
void       Forward();
void       Assume();
void       AssmPrt();
void       fatal();

VALUE      stubINP();
VALUE      stubINV();
VALUE      stubAND();
VALUE      stubOR();
VALUE      stubNAND();
VALUE      stubNOR();
VALUE      stubXOR();


#endif
```

C.3 ADDINPUT.C

```
/*===================================================================*/
/*                                                                   */
/* AddInputInfo -- determine the static connectivity information     */
/*                                                                   */
/* The AddInputInfo procedure is used to determine the static        */
/* connectivity information.  It performs a recursive depth-first    */
/* traversal of the circuit web data struture for eech input.        */
/* Gates that are visited during the traversal for input 1 are       */
/* dependent on input 1.  A bit string kept for each gate is used    */
/* to record which inputs that gate depends on.                      */
/*                                                                   */
/*                                                                   */
/* Copyright (c) 1988, 1989                                          */
/*          Neil Erdwien end Kansas State University                 */
/*          All rights reserved.                                     */
/*                                                                   */
/*===================================================================*/


/*===================================== Standard C include files */
#ifdef ANSI
#include <stddef.h>
#include <stdlib.h>
#endif
#include <stdio.h>
#include <malloc.h>
#include <string.h>


/*=============================== Program-specific include files */
#include "symsim.h"
#include "bitstrin.h"


/*================================= Private function prototypes */
#ifdef ANS1
stetic void InputTraverse(int bitnum, GBLOK *gate);
#endif


static
void
InputTraverse(bitnum, gate)

int bitnum;
GBLOK *gate;

{

    GLUE  *FenoutPtr;

    /* The portion of the circuit web rooted at a given input is  */
    /* not a tree because it may have cycles.  The VISITED field  */
    /* and the global VisitedValue are used to detect and truncate */
    /* loops.                                                     */
    if (gate->visited != VisitedValue) {
```

```
        /* This node hasn't been visited yet...                    */
        gate->visited = VisitedValue;

        BitSet(gate->causes, bitnum);

        /* Follow the fan-out information to recurse down the tree. */
        for (FanoutPtr = gate->olist;
             FanoutPtr != NULL;
             FanoutPtr = FanoutPtr->next) {

            InputTraverse(bitnum, FanoutPtr->gate);
        }
    }
}


void
AddInputInfo(Iptr)

GBLOK *Iptr[];

{
    int i;

    /* Perform a recursive depth-first traversal for each input.   */
    for (i = 0; Iptr[i] != NULL; i++) {
        VisitedValue++;
        InputTraverse(i, Iptr[i]);
    }

}
```

C.4   ASSUME.C

```
/*==================================================================*/
/*                                                                  */
/* assume -- record an assumption about a node value                */
/*                                                                  */
/* This function is called to make an assumption about the output   */
/* value of a node.  The assumption is recorded and the             */
/* simulation proceeds forward with the new information.            */
/*                                                                  */
/* The assumption is recorded so that it can be removed later.      */
/*                                                                  */
/* Assume is called with two arguments -- a pointer to a gate in    */
/* the circuit web and e sequence value.  The value of the output   */
/* of the indicated gate is set to the sequence value.              */
/*                                                                  */
/* Assume implements a recursive depth-first traversal of the       */
/* search tree.  The initial call to assume is from the main        */
/* routine; the subsequent calls are recursive calls from assume    */
/* itself.                                                          */
/*                                                                  */
/*                                                                  */
/* Copyright (c) 1988, 1989                                         */
```

- 76 -

```
/*          Neil Erdwien and Kansas State University          */
/*          All rights reserved.                              */
/*                                                            */
/*================================================================*/


/*===================================== Standard C include files */
#ifdef ANSI
#include <stddef.h>
#include <stdlib.h>
#endif
#include <stdio.h>
#include <malloc.h>
#include <assert.h>


/*============================== Program-specific include files */
#include "symsim.h"
#include "bitstrin.h"


/*==================================== Global external variables */
int       HaveTAassumption = 0;
TIME      TAmin, TAmax;


void
Assume(gate, seq)

GBLOK    *gate;
SEQUENCE seq;

{
   GLUE   *FanoutPtr;
   int    i, j;
   int    lmax, lcountMax;
   int    lcount[MAXINPUTS];
   SEQUENCE tempseq;

   if (msglevel >= 6) {
      printf("Assumption %d: %s gate %s is set to %s\n", Acount,
             Func2Str(gate->function), gate->name, Seq2Str(seq));
   }

   push(gate);
   Acount++;

   /* Set the gate to the assumed value and move forward.       */
   gate->seq = seq;

   if (gate->seq.val != unk) {
      for (FanoutPtr = gate->olist;
           FanoutPtr != NULL; FanoutPtr = FanoutPtr->next) {
         Forward(FanoutPtr->gate);
      }
   }
```

- 77 -

```c
/* If we haven't reached the maximum assumption count, we      */
/* look for an node to make an assumption about.  As a         */
/* heuristic, we go through all the dead-ends that are         */
/* still unknown, and make a tally of which inputs can         */
/* affect each dead-end.  The assumption is made about the     */
/* input that affects the most dead-ends.                      */

if (Acount <= Amax) {

    for (i = 0; i < MAXINPUTS; i++)
        Icount[i] = 0;

    for (i = StackPtr-1; i >= 0; i--) {
        gate = Stack[i];
        assert(gate != NULL);
        if (gate->seq.val == unk) {

            for (j = 0; Iptr[j] != NULL; j++) {
                if (BitTest(gate->causes, j))
                    Icount[j]++;
            }
        }
        if (gate->function == input)
            break;
    }

    /* Find the input that is 'unknown' with the largest count. */
    Imax = -1;
    IcountMax = -1;
    for (i = 0; Iptr[i] != NULL; i++) {
        if (Icount[i] > IcountMax  &&  Iptr[i]->seq.val == unk) {
            IcountMax = Icount[i];
            Imax = i;
        }
    }

    if (IcountMax > 0) {

        tempseq.t1  = 0;
        tempseq.t2  = 10;
        tempseq.tA  = 0;
        tempseq.val = zero;
        Assume(Iptr[Imax], tempseq);
        tempseq.val = one;
        Assume(Iptr[Imax], tempseq);
        if (HaveTAassumption == 0) {
            HaveTAassumption = 1;
            TAmin = 0;
            TAmax = MAXTIME;
            tempseq.val = up;
            tempseq.tA  = 1;
            Assume(Iptr[Imax], tempseq);
            tempseq.val = down;
            Assume(Iptr[Imax], tempseq);
            HaveTAassumption = 0;
        }
    }
}
```

```
    }

    /* Pop the stack until the previous assumption is found.        */
    /* Each gate stacked is set back to 'unknown'.                  */
    while(1) {
        gate = pop;
        assert(gate != NULL);
        gate->seq.val = unk;
        gate->seq.tA  = 0;
        if (gate->function == input)
            break;
    }

    Acount--;

    if (msglevel >= 6) {
        printf("Done with assumption %d\n\n", Acount);
    }

}
```

C.5   BITSTRIN.H

```
/*===================================================================*/
/*                                                                   */
/* BITSTRING -- set and test bits in a bit string                    */
/*                                                                   */
/* This header file defines macros to aid in the manipulation of     */
/* bit strings.  A bit string is implemented as an array of          */
/* short integers.  The macros below can set, reset, and test any    */
/* bit in the string.                                                */
/*                                                                   */
/*                                                                   */
/* Copyright (c) 1988, 1989                                          */
/*         Neil Erdwien and Kansas State University                  */
/*         All rights reserved.                                      */
/*                                                                   */
/*===================================================================*/


/*================================= Manifest constant definitions */
#define BitsPerShort 16


#define BitWord(N)           ( (N) / BitsPerShort )
#define BitMask(N)           ( 1 << ((N) % BitsPerShort) )
#define BitSet(string, N)    ( (string) [BitWord(N)] |=  BitMask(N) )
#define BitReset(string, N)  ( (string) [BitWord(N)] &= ~BitMask(N) )
#define BitTest(string, N)   ( (string) [BitWord(N)] &   BitMask(N) )
```

- 79 -

C.6   FATAL.C

```
/*=====================================================================*/
/*                                                                     */
/* fatal -- issue a fatal error message and abort                      */
/*                                                                     */
/* This function is called when a non-recoverable error occurs.        */
/* A message is issued to stderr, and the program is aborted.          */
/*                                                                     */
/* This function uses the variable argument facilities available       */
/* in AT&T System V's C compiler rather than ANSI C.  This was         */
/* done because the System V facilities are more universal.  This      */
/* method has been tested and works on the following platforms:        */
/*                                                                     */
/*     MS-DOS Microsoft C version 5.1                                  */
/*     AT&T Unix System V                                              */
/*     SUNOS version 4.0                                               */
/*                                                                     */
/*                                                                     */
/* Calling sequence:                                                   */
/*                                                                     */
/*     fatal(format [,argument]...);                                   */
/*                                                                     */
/* fatal is called just like printf, i.e., with a format string        */
/* and a variable arguement list of values to be substituted into      */
/* the message.  All of the message editing facilities of printf       */
/* are available.                                                      */
/*                                                                     */
/* The message should be as domain-specific as possible.  If a         */
/* system run-time error has also occurred, the message from           */
/* perror will be printed after the message passed to fatal.           */
/* This message usually indicates *WHY* an error occurred -- your       */
/* message should describe the error in application terminology.        */
/* Your message should not end in a newline.                           */
/*                                                                     */
/* For example, to use fatal in a file-opening context:                */
/*                                                                     */
/*     fp = fopen(TimeName, "r");                                      */
/*     if (fp == NULL)                                                 */
/*         fatal("Error opening time file '%s'", TimeName);            */
/*                                                                     */
/* If an error does occur, the message printed will be:                */
/*                                                                     */
/* Error opening time file 'whatever': No such file or directory       */
/*                                                                     */
/* It is very good style to include information such as the fact        */
/* that the "time" file was the one having trouble, as well as         */
/* the actual file name.                                               */
/*                                                                     */
/*                                                                     */
/* Copyright (c) 1988, 1989                                            */
/*          Neil Erdwien and Kansas State University                   */
/*          All rights reserved.                                       */
/*                                                                     */
/*=====================================================================*/


/*=================================== Standard C include files */
```

- 80 -

```
#ifdef ANSI
#include <stddef.h>
#include <stdlib.h>
#endif
#include <stdio.h>
#include <varargs.h>

/* The Unix systems have ERRNO in a separate header file.        */
#ifndef MSDOS
#include <errno.h>
#endif


/*================================== Program-specific include files */
#include "symsim.h"


void
fatal(va_alist)

va_dcl

{
    char    *fmt;
    va_list arg_ptr;
    char    buffer[128];

    fputc('\n', stderr);

    va_start(arg_ptr);

    /* The first argument is a character string used as a format  */
    /* list like printf's.                                        */
    fmt = va_arg(arg_ptr, char *);

    /* Arg_ptr now points at the second (and following) parameters */
    /* in a format that can be passed to vsprintf directly.        */
    vsprintf(buffer, fmt, arg_ptr);
    va_end(arg_ptr);

    /* Buffer is now filled with our error message -- print it.   */
    /* If a run-time system error has also occurred, the perror   */
    /* function is used to print both messages.  Otherwise, the   */
    /* messages is printed diractly.                              */
    if (errno)
       perror(buffer);
    else {
       fputs(buffer, stderr);
       fputc('\n', stderr);
    }

    /* Stop the program rather than return to the caller.         */
    exit(1);

}
```

- 81 -

C.7   FORWARD.C

```
/*===================================================================*/
/*                                                                   */
/* Forward -- propagate logic values forward through the circuit     */
/*                                                                   */
/* Forward is tha basic simulation mechanism of SYMSIM.  Forward     */
/* is passed a pointer to a gate in the circuit web.  That gate      */
/* is simulated, and if the result of the simulation is any value    */
/* except unknown, Forward is called recursively for every gate      */
/* in the simulated gate's fanout.                                   */
/*                                                                   */
/* Forward thus performs a recursive depth-first traversal of tha    */
/* circuit web.  When Forward returns from a call, the result of     */
/* simulating the indicated gate are propagated as far as            */
/* possible.                                                         */
/*                                                                   */
/* The results of simulation, whether unknown or not, are pushed     */
/* onto the stack so that the consequences of an assumption can      */
/* be removed.                                                       */
/*                                                                   */
/*                                                                   */
/* Copyright (c) 1988, 1989                                          */
/*          Neil Erdwien and Kansas State University                 */
/*          All rights reserved.                                     */
/*                                                                   */
/*===================================================================*/


/*======================================= Standard C include files */
#ifdef ANSI
#include <stddef.h>
#include <stdlib.h>
#endif
#include <stdio.h>
#include <math.h>
#include <assert.h>
#include <malloc.h>
#include <string.h>


/*================================= Program-specific include files */
#include "symsim.h"


/*===================================== Global external variables */
GBLOK    *Stack[STACKSIZE];
int      StackPtr = 0;


void
Forward(gate)

GBLOK *gate;

{

   GLUE  *FanoutPtr;
```

- 82 -

```
    if (msglevel >= 8) {
        printf("    simulating %s gate %s\n", Func2Str(gate->function),
               gate->name);
    }

    Sim1(gate);
    push(gate);

    if (msglevel >= 8) {
        printf("    ...result: %s\n", Seq2Str(gate->seq));
    }

    /* The results of the simulation are known.  Either the output */
    /* is now known (and we can move forward) or the output is     */
    /* unknown (and we have encountered a dead end.                */
    if (gate->seq.val != unk) {

        for (FanoutPtr = gate->olist;
             FanoutPtr != NULL;
             FanoutPtr = FanoutPtr->next) {

            /* Check to see if the output of this gate is known.   */
            /* If so, this gate has already been simulated and we  */
            /* can trim this part of the tree.  Otherwise, infinite */
            /* loops could result from cyclic circuit webs.        */
            if (FanoutPtr->gate->seq.val == unk)
                Forward(FanoutPtr->gate);
        }

    }

}
```

C.8   FUNC2STR.C

```
/*===================================================================*/
/*                                                                   */
/* Func2Str -- convert a function type into a character string       */
/* Str2Func -- convert a character string into a function type       */
/*                                                                   */
/* These functions convert between variables of type GTYPE and       */
/* printable character strings meant for human reading.              */
/*                                                                   */
/* For Func2Str, if an invalid GTYPE value is sent in, the string    */
/* "[invalid]" is returned.  For Str2Func, an unrecognized string    */
/* results in the "invalid" GTYPE value being returned.              */
/*                                                                   */
/* Func2Str returns a pointer to the character string                */
/* representation and Str2Func returns the actual GTYPE value.       */
/*                                                                   */
/* Func2Str returns the address of an internal static buffer and     */
/* hence the value returned must be used before the next call.       */
/* This causes problems when printing two values in a single         */
/* printf call -- printf calls this function twice and then          */
/* prints the result.  The solution is to use separate printf        */
```

```
/* calls.                                                              */
/*                                                                     */
/*                                                                     */
/* Copyright (c) 1988, 1989                                            */
/*          Neil Erdwien and Kansas State University                   */
/*          All rights reserved.                                       */
/*                                                                     */
/*===================================================================*/


/*====================================== Standard C include files */
#ifdef ANSI
#include <stddef.h>
#include <stdlib.h>
#endif
#include <stdio.h>
#include <string.h>


/*================================= Program-specific include files */
#include "symsim.h"


/*======================================= Private static variables */
static char *names[MAXGTYPES] = {
    "Input",
    "Output",
    "Inverter",
    "AND",
    "OR",
    "NAND",
    "NOR",
    "XOR",
    "[invalid]"
};


char *
Func2Str(function)

GTYPE function;

{
    int i;

    i = (int) function;  /* Convert the value to a subscript...   */
                         /* ...ensure that it is in range...      */
    if (i < 0  ||  i >= MAXGTYPES)
        i = MAXGTYPES-1;

    return names[i];     /* ...and return the corresponding entry. */
}


GTYPE
Str2Func(name)

char *name;
```

- 84 -

```
{
    int i;

    for (i = 0; i < MAXGTYPES-1; i++) {
        if (stricmp(name, names[i]) == 0) {
            return (GTYPE) i;
        }
    }

    return invalid;
}
```

C.9   GETCIRC.C

```
/*==================================================================*/
/*                                                                  */
/* GetCircuit -- read a Triviel-Circuit-Format file                 */
/*                                                                  */
/* This function reads an input file in Trivial Circuit Format      */
/* end creates e circuit web dete structure.                        */
/*                                                                  */
/* A two pess method is used for convenience.  This allows gate     */
/* names to be used before they ere defined.                        */
/*                                                                  */
/* The first pass creates a node for each circuit element but       */
/* ignores connection information because a connection may be       */
/* requested to a not-yet-defined gate.                             */
/*                                                                  */
/* The second pass makes the connections between the nodes          */
/* creeted in pass 1.  This is easy because all gates have been     */
/* defined.                                                         */
/*                                                                  */
/* The gate names are managed with a separate teble packege to      */
/* perform the character-string-name to pointer-to-deta-structure   */
/* mapping.                                                         */
/*                                                                  */
/* Error situations are diagnosed, but the error messages could     */
/* be improved to aid in the analysis of the error.                 */
/*                                                                  */
/*                                                                  */
/* Copyright (c) 1988, 1989                                         */
/*           Neil Erdwien and Kansas State University               */
/*           All rights reserved.                                   */
/*                                                                  */
/*==================================================================*/


/*===================================== Standerd C include files */
#ifdef ANS1
#include <stddef.h>
#include <stdlib.h>
#endif
#include <stdio.h>
#include <malloc.h>
#include <string.h>
```

- 85 -

```
/*================================== Program-specific include files */
#include "symsim.h"
#include "table.h"


/*==================================== Private function prototypes */
#ifdef ANS1
static long GetTPass1(F1LE *fp, TABLE *tptr,
                      GBLOK *lptr[], GBLOK *Optr[]);
static long GetTPass2(F1LE *fp, TABLE *tptr);
#endif


static
long
GetTPass1(fp, tptr, lptr, Optr)

F1LE *fp;
TABLE *tptr;
GBLOK *lptr[];
GBLOK *Optr[];

{
   long   count = 0;
   char   *token;
   GBLOK  *gate;
   int    result;
   int    InputCounter = 0;
   int    OutputCounter = 0;

   token = GetToken(fp);
   while (stricmp("ELEMENT", token) == 0) {

      token = GetToken(fp);              /* Read the name              */
      if (feof(fp))
         break;

      gate = (GBLOK *) malloc(sizeof(GBLOK));
      if (gate==NULL)
         fatal("Error allocating a GBLOK for element '%s'", token);
      gate->seq.val   = unk;
      gate->seq.t1    = 0;
      gate->seq.t2    = 0;
      gate->seq.tA    = 0;
      gate->visited   = 0;
      gate->causes[0] = 0x0000;
      gate->causes[1] = 0x0000;
      gate->causes[2] = 0x0000;
      gate->causes[3] = 0x0000;
      gate->olist     = NULL;
      gate->ilist     = NULL;

      gate->name = strdup(token);
      result = TableAdd(tptr, gate->name, gate);
      if (result)
         fatal("Duplicate name '%s' in circuit file", gate->name);
```

- 86 -

```
        token = GetToken(fp);          /* Read the element type      */
        if (feof(fp))
            break;
        gate->function  = Str2Func(token);
        if (gate->function == invalid)
            fatal("Unknown function type '%s'", token);
        if (gate->function == input) {
            Iptr[InputCounter++] = gate;
        }
        if (gate->function == output) {
            Optr[OutputCounter++] = gate;
        }

        count++;

        /* Skip all the connection information                        */
        while (1) {
            token = GetToken(fp);
            if (feof(fp) || stricmp("ELEMENT", token)==0)
                break;
        }

    }

    if (!feof(fp)) {
        fatal("Syntax error");
    }

    return count;

}


static
long
GetTPass2(fp, tptr)

FILE *fp;
TABLE *tptr;

{
    long  count = 0;
    char  *token;
    GBLOK *gate;
    GBLOK *g2;
    GLUE  *glueptr;

    token = GetToken(fp);
    while (stricmp("ELEMENT", token) == 0) {

        token = GetToken(fp);          /* Read the name              */
        gate = (GBLOK *) TableLookup(tptr, token);
        if (gate == NULL)
            fatal("Circuit element '%s' is not defined", token);
        token = GetToken(fp);          /* Read the element type       */

        while (1) {
            token = GetToken(fp);
```

- 87 -

```
        if (feof(fp) || stricmp("ELEMENT", token)==0)
            break;

        g2 = (GBLOK *) TableLookup(tptr, token);
        if (g2 == NULL)
            fatal("Circuit element '%s' is not defined", token);

        /* Tie the output of 'gate' to the input of 'g2'.        */
        glueptr = (GLUE *) malloc(sizeof(GLUE));
        if (glueptr == NULL)
            fatal("Error allocating GLUE");
        glueptr->next = gate->olist;
        gate->olist = glueptr;
        glueptr->gate = g2;

        /* Tie the input of 'g2' to the output of 'gate'.        */
        glueptr = (GLUE *) malloc(sizeof(GLUE));
        if (glueptr == NULL)
            fatal("Error allocating GLUE");
        glueptr->next = g2->ilist;
        g2->ilist = glueptr;
        glueptr->gate = gate;

        count++;
    }

    }

    if (!feof(fp)) {
        fatal("Syntax error");
    }

    return count;

}


int
GetCircuit(filename, lptr, Optr)

char *filename;
GBLOK *Iptr[];
GBLOK *Optr[];

{

    FILE  *fp;
    TABLE *tptr;
    long  count;
    int   i;

    fp = fopen(filename, "r");
    if (fp == NULL) {
        fatal("Error opening file '%s'", filename);
    }

    tptr = TableCreate();
```

```
    for (i = 0; i < MAXINPUTS; i++) {
        Iptr[i] = NULL;
    }

    for (i = 0; i < MAXOUTPUTS; i++) {
        Optr[i] = NULL;
    }

    count = GetTPass1(fp, tptr, Iptr, Optr);
    if (msglevel >= 2)
        printf("   %ld circuit elements created\n", count);

    rewind(fp);
    count = GetTPass2(fp, tptr);
    if (msglevel >= 2)
        printf("   %ld connections created\n\n", count);

    TableFree(tptr);

    fclose(fp);

    return 0;
}


/* AT&T's Unix System V doesn't include a case-insignificant    */
/* string compare function.   The following code implements one. */

#ifdef SYSV

#include <ctype.h>

int stricmp(a, b)

char *a, *b;

{

    while (*a && *b) {
        if (tolower(*a) > tolower(*b))
            return 1;
        if (tolower(*a) < tolower(*b))
            return -1;

        a++;
        b++;
    }

    if (*a)
        return 1;

    if (*b)
        return -1;

    return 0;

}
```

- 89 -

```
#endif
```

C.10   GETOPT.C

```c
/*====================================================================*/
/*                                                                    */
/* getopt -- standard Unix System V command line option parsing       */
/*                                                                    */
/* This function is my version of the  getopt  function that is       */
/* standard on many Unix systems, particularly AT&T's System V.       */
/*                                                                    */
/* See the Unix documentation for details on how to use this          */
/* function.                                                          */
/*                                                                    */
/* This function is designed for MS-DOS use.  This function is        */
/* not needed on most Unix systems because they supply a version      */
/* of getopt.                                                         */
/*                                                                    */
/* This version of  getopt  is based on the version placed in the     */
/* public domain by AT&T at the 1985 UNIFORUM conference.  The        */
/* original is available in volume 3 of the comp.sources.unix         */
/* newsgroup.                                                         */
/*                                                                    */
/* The following changes have been made:                              */
/*                                                                    */
/* 1.   The use of a literal 2 for  stderr  in the   write   call     */
/*      has been replaced by the use of  fputs  and  fputc .          */
/* 2.   Tests for string comparisons returning NULL now test for      */
/*      0.                                                            */
/* 3.   The  getopt  function itself is now declared with             */
/*      an ANSI C prototype.                                          */
/* 4.   General clean-up and commenting.                              */
/*                                                                    */
/*                                                                    */
/* Copyright (c) 1988, 1989                                           */
/*          Neil Erdwien and Kansas State University                  */
/*          All rights reserved.                                      */
/*                                                                    */
/*====================================================================*/

/*===================================== Standard C include files */
#include <stdio.h>
#include <strings.h>


/*===================================== Global external variables */
int    opterr = 1;        /* If set to 0 no messages will be printed  */
int    optind = 1;        /* Index to the string being processed      */
int    optopt;            /* Option character being processed         */
char   *optarg;           /* Pointer to the option argument           */


#define ERR(s, c) if(opterr){\
   fputs(argv[0], stderr);\
   fputs(s      , stderr);\
```

- 90 -

```
    fputc(c        , stderr);\
    fputc('\n'     , stderr);}


int
getopt(int argc, char **argv, char *opts)

{

    static   int sp = 1;     /* Index into the string being processed */
    register int c;          /* Option character being processed      */
    register char *cp;       /* Pointer to the match within  *opts    */

    if (sp == 1) {

        /* This is the start of a new option string               */
                              /* If we've done all the strings...   */
        if (optind >= argc ||
                              /* ...or it doesn't start with '-'...*/
            argv[optind][0] != '-' ||
                              /* ...or it is only 1 char long...    */
            argv[optind][1] == '\0')
                              /* then return end-of-options value. */
            return EOF;

        /* Check for an explicit "--" to indicate end-of-options.  */
        if (strcmp(argv[optind], "--") == 0) {
            /* Yes, "--" was found.                                */
            /* Skip this option and return end-of-opt             */
            optind++;
            return EOF;
        }
    }

    /* Get the next option letter                              */
    optopt = c = argv[optind][sp];

    if (c == ':'   ||                  /* If the character is an ':'... */
                                       /* ...or isn't in the option str. */
        (cp=strchr(opts, c)) == 0) {
                                       /* ...then report an error.       */
        ERR(": illegal option -- ", c);
        if (argv[optind][++sp] == '\0') {
            optind++;
            sp = 1;
        }
        return '?';
    }

    /* Check to see if this option needs an argument           */
    if (*++cp == ':') {
        /* Yes, an argument is needed                          */

        /* If there is something left in this string, assume it is */
        /* the argument for this option.                           */
        if (argv[optind][sp+1] != '\0')
            optarg = &argv[optind++][sp+1];
        else if (++optind >= argc) {
```

```
                                   /* Otherwise use the next string. */
        ERR(": option requires an argument -- ", c);
        sp = 1;
        return '?';
    } else
        optarg = ergv[optind++];
    sp = 1;

} else {
    /* No argument is needed for this option                         */

    /* If there is nothing left, skip to the next string.            */
    if (ergv[optind][++sp] == '\0') {
        sp = 1;
        optind++;
    }
    optarg = NULL;                  /* Emphasize there is no arg     */
}

return c;

}
```

C.11   LONGSIM1.C

```
/*=====================================================================*/
/*                                                                     */
/* Siml -- simulate a single logic element                             */
/*                                                                     */
/* This function takes a single argument, a pointer to the GBLOK       */
/* representing the gate to be simulated.  The input values to         */
/* the gate ere found by traversing the circuit web.                   */
/*                                                                     */
/* If any outputs of the circuit element can be determined, the        */
/* values in the circuit web ere updated.                              */
/*                                                                     */
/* There are two files that both contain functions named Siml.         */
/* This file, LONGSIM1, uses the 5-level sequence to calculete         */
/* the result.  This version of Siml is used by TTABLE to              */
/* calculate the tables.  SYMSIM uses the tables thus created and      */
/* a different version of Siml.                                        */
/*                                                                     */
/* At one time, LONGSIM1 and TSIM1 were interchangeable by simply      */
/* linking one or the other.  TSIM1 has since been enhanced end        */
/* must be used by SYMSIM.  Conversely, TTABLE must use LONGSIM1.      */
/*                                                                     */
/*                                                                     */
/* Copyright (c) 1988, 1989                                            */
/*          Neil Erdwien and Kansas State University                   */
/*          All rights reserved.                                       */
/*                                                                     */
/*=====================================================================*/


/*==================================== Standard C include files */
```

- 92 -

```
#ifdef ANSI
#include <stddef.h>
#include <stdlib.h>
#endif
#include <stdio.h>
#include <math.h>
#include <assert.h>
#include <malloc.h>
#include <string.h>

/* Unix doesn't define size_t.                                    */
#ifndef MSDOS
typedef unsigned int size_t;
#endif


/*================================= Program-specific include files */
#include "symsim.h"


/*=================================== Manifest constant definitions */
/* MAXEVENTS is the maximum number of events per gate.  It should */
/* be set to at least 4*(number of inputs).  Because this is one  */
/* of the most-often executed parts, no out-of-bounds checking is */
/* performed.  The number used as the number of inputs is large   */
/* to compensate.                                                 */
#define MAXEVENTS (4*MAXINPUTS)


/*====================================== Global external variables */
extern    TIME   tPLHmin[MAXGTYPES];
extern    TIME   tPLHmax[MAXGTYPES];
extern    TIME   tPHLmin[MAXGTYPES];
extern    TIME   tPHLmax[MAXGTYPES];

#ifdef ANSI
VALUE     (*stubs[]) ( VALUE inputs[], int numinputs );
#else
VALUE     (*stubs[]) ( );
#endif


/*====================================== Private static variables */

typedef struct event {
    int      gate;
    TIME     time;
    VALUE    val;
} EVENT;

EVENT     event[MAXEVENTS];
int       numevents;

VALUE     inputs[MAXINPUTS];
int       numinputs;
SEQUENCE  Oevents[MAXEVENTS];
int       numout;
```

- 93 -

```
/*==================================== Private function prototypes */
#ifdef ANSI
static void CollectInputEvents(GBLOK *gate);
static int timecompare(EVENT *a, EVENT *b);
static SEQUENCE CollapseSequence(SEQUENCE Oevents[], int numout);
static SEQUENCE AdjustTime(SEQUENCE seq, GTYPE function);
#endif


/*===================================================================*/
/*                                                                   */
/* CollectInputEvents -- create events based on a gate's inputs      */
/*                                                                   */
/* This function examines each input to the indicated gate.  Each    */
/* 9-level value is turned into a sequence of "events" or 5-level    */
/* values tagged with a time value.                                  */
/*                                                                   */
/*===================================================================*/

static
void
CollectInputEvents(gate)

GBLOK *gate;

{
    GLUE    *pi;

    numevents = 0;
    numinputs = 0;
    for (pi = gate->ilist; pi != NULL; pi = pi->next) {
        numinputs++;

        switch(pi->gate->seq.val) {

        case zero:
        case one:
            event[numevents].gate = numinputs;
            event[numevents].time = INITIAL;
            event[numevents].val  = pi->gate->seq.val;
            numevents++;
            break;

        case up:
            event[numevents].gate = numinputs;
            event[numevents].time = INITIAL;
            event[numevents].val  = zero;
            numevents++;
            event[numevents].gate = numinputs;
            event[numevents].time = pi->gate->seq.t1;
            event[numevents].val  = up;
            numevents++;
            event[numevents].gate = numinputs;
            event[numevents].time = pi->gate->seq.t2;
            event[numevents].val  = one;
            numevents++;
            break;
```

- 94 -

```
case down:
   event[numevents].time = INITIAL;
   event[numevents].gate = numinputs;
   event[numevents].val  = one;
   numevents++;
   event[numevents].time = pi->gate->seq.t1;
   event[numevents].gate = numinputs;
   event[numevents].val  = down;
   numevents++;
   event[numevents].time = pi->gate->seq.t2;
   event[numevents].gate = numinputs;
   event[numevents].val  = zero;
   numevents++;
   break;

case st0:
   event[numevents].time  = INITIAL;
   event[numevents].gate = numinputs;
   event[numevents].val   = zero;
   numevents++;
   event[numevents].time  = pi->gate->seq.t1;
   event[numevents].gate = numinputs;
   event[numevents].val   = unk;
   numevents++;
   event[numevents].time  = pi->gate->seq.t2;
   event[numevents].gate = numinputs;
   event[numevents].val   = zero;
   numevents++;
   break;

case st1:
   event[numevents].time  = INITIAL;
   event[numevents].gate = numinputs;
   event[numevents].val   = one;
   numevents++;
   event[numevents].time  = pi->gate->seq.t1;
   event[numevents].gate = numinputs;
   event[numevents].val   = unk;
   numevents++;
   event[numevents].time  = pi->gate->seq.t2;
   event[numevents].gate = numinputs;
   event[numevents].val   = one;
   numevents++;
   break;

case dy1:
   event[numevents].time  = INITIAL;
   event[numevents].gate = numinputs;
   event[numevents].val   = zero;
   numevents++;
   event[numevents].time  = pi->gate->seq.t1;
   event[numevents].gate = numinputs;
   event[numevents].val   = unk;
   numevents++;
   event[numevents].time  = pi->gate->seq.t2;
   event[numevents].gate = numinputs;
   event[numevents].val   = one;
   numevents++;
```

```
          break;

       case dy0:
          event[numevents].time  = INITIAL;
          event[numevents].gate = numinputs;
          event[numevents].val    = one;
          numevents++;
          event[numevents].time  = pi->gate->seq.t1;
          event[numevents].gate = numinputs;
          event[numevents].val    = unk;
          numevents++;
          event[numevents].time  = pi->gate->seq.t2;
          event[numevents].gate = numinputs;
          event[numevents].val    = zero;
          numevents++;
          break;

       case unk:
          event[numevents].time  = INITIAL;
          event[numevents].gate = numinputs;
          event[numevents].val    = unk;
          numevents++;
          break;

       } /* switch(pi->gate->seq.val) */
    }
}


/*===================================================================*/
/*                                                                   */
/* timecompare -- compare two time values                            */
/*                                                                   */
/* The following function is needed to compare two time values.      */
/* It is passed to the C library's qsort routine to sort the         */
/* input events into ascending order.                                */
/*                                                                   */
/*===================================================================*/

static
int
timecompare(a, b)

EVENT *a;
EVENT *b;

{
    return a->time - b->time;
}


/*===================================================================*/
/*                                                                   */
/* CollapseSequence -- convert a 5-level sequence into a 9-level     */
/*                     value                                         */
/*                                                                   */
/*===================================================================*/
```

```
static
SEQUENCE
CollapseSequence(Oevents, numout)

SEQUENCE Oevents[];
int numout;

{
    int      i;
    VALUE    pre, post;
    int      first, last;
    SEQUENCE result;

    if (msglevel >= 9) {
        printf("\nAfter simulation, collapsing %d events:\n", numout);
        for (i=0; i<numout; i++) {
            printf("  %s", Seq2Str(Oevents[i]));
        }
        printf("\n");
    }

    /*-------------------------------------------------------------*/
    /* The sequence of output values is collected into the Oevents */
    /* array.                                                      */

    assert(numout>0);
    for (i=0; i<numout; i++)
        assert(Oevents[i].val >= zero  &&  Oevents[i].val <= unk);


    /*-------------------------------------------------------------*/
    /* Skip over any equal values at the start of the sequence     */
    /*-------------------------------------------------------------*/

    pre = Oevents[0].val;
    for (first=0; pre==Oevents[first].val && first<numout; first++)
        ;

    if (first == numout) {        /* The whole sequence is one value */
        result.val = pre;
        result.t1  = Oevents[0].t1;
        result.t2  = Oevents[numout-1].t1;
        return result;
    }

    /* FIRST points at the first different value in the array...   */
    first--;                      /* ...make it the last equal value */

    if (pre == unk) {             /* The first value is unk...      */
        result.val = unk;         /* ...the whole sequence must be  */
        result.t1  = Oevents[0].t1;
        result.t2  = Oevents[numout-1].t1;
        return result;
    }


    /*-------------------------------------------------------------*/
    /* Now do the same for the end of the sequence                 */
```

- 97 -

```c
/*--------------------------------------------------------------*/
post = Oevents[numout-1].val;
for (last=numout-1; post==Oevents[last].val && last>0; last--)
    ;

assert(last>=first);    /* Should never go over the whole seq.  */

/* LAST points at the last different value in the array...       */
last++;                 /* ...make it the first equal value      */

if (post == unk) {          /* The first value is unk...         */
    result.val = unk;       /* ...the whole sequence must be     */
    result.t1  = Oevents[0].t1;
    result.t2  = Oevents[numout-1].t1;
    return result;
}


/*--------------------------------------------------------------*/
/* The interesting part of the sequence is bracketed by         */
/* [FIRST,LAST] inclusive.  The first value is in PRE,          */
/* the last value in POST.                                      */
/*                                                              */
/*      A   A   A   A   X  X  X   B   B   B   B                 */
/*                      |         |                            */
/*      FIRST -------/              \------ LAST               */
/*--------------------------------------------------------------*/
result.t1  = Oevents[ first+1 ].t1;
result.t2  = Oevents[ last    ].t1;

if (pre==post && (pre==zero || pre==one)) {
    if (pre==zero)
        result.val = st0;
    else
        result.val = st1;
    return result;
}

if (pre  == zero   && post == one) {

    result.val = up;    /* Assume this is just a 0-1 transition  */

    /* Loop over the intermediate states looking for            */
    /* non UP values.                                           */
    for (i = first+1; i<last; i++) {
        if (Oevents[i].val != up) {
            /* Found one -- must be a dynamic hazard            */
            result.val = dy1;
            break;
        }
    }
} else {

    result.val = down; /* Assume this is just a 1-0 transition   */

    /* Loop over the intermediate states looking for            */
```

```
      /* non DOWN values.                                                 */
      for (i = first+1; i<last; i++) {
          if (Oevents[i].val != down) {
              /* Found one -- must be a dynamic hazard                     */
              result.val = dy0;
              break;
          }
      }
  }

  return result;
}


/*===================================================================*/
/*                                                                   */
/* AdjustTime -- adjust time values for the delay of this element */
/*                                                                   */
/* The AdjustTime function alters the time values of a sequence      */
/* to include the propagation delay of the element being             */
/* simulated.                                                        */
/*                                                                   */
/*===================================================================*/

static
SEQUENCE
AdjustTime(seq, function)

SEQUENCE seq;
GTYPE function;

{

    switch (seq.vel) {
        case zero:
        case one:
        case unk:
          break;
        case up:
        cese dy0:
          if (seq.t1 == -1)
              seq.t1 = 0;
              seq.t1 = seq.t1 + tPLHmin[function];
          seq.t2 = seq.t2 + tPLHmax[function];
          break;
        case down:
        case dy1:
          if (seq.t1 == -1)
              seq.t1 = 0;
              seq.t1 = seq.t1 + tPHLmin[function];
          seq.t2 = seq.t2 + tPHLmax[function];
          break;
        case st0:
          if (seq.t1 == -1)
              seq.t1 = 0;
              seq.t1 = seq.t1 + tPLHmin[function];
          seq.t2 = seq.t2 + tPHLmax[function];
          break;
```

```
        case  stl:
            if (seq.tl == -1)
                seq.tl = 0;
                seq.tl = seq.tl + tPHLmin[function];
            seq.t2 = seq.t2 + tPLHmax[function];
            break;
        default:
            fatal("Unknown logic value in AdjustTime");
    }

    return seq;

}


void
Siml(gate)

GBLOK *gate;

{

    int     i;
#ifdef ANSI
    VALUE   (*stub)(VALUE inputs[], int numinputs);
#else
    VALUE   (*stub)();
#endif
    SEQUENCE seq;
    TIME     prevtime;

    if (gate->function == input) {
        return;
    }


    /*------------------------------------------------------------------*/
    /* First, collect the gate "events" from the various inputs      */
    /*------------------------------------------------------------------*/

    CollectInputEvents(gate);

    /* Sort the events into chronological order                    */
    qsort(event, (size_t) numevents, sizeof(EVENT), timecompare);

    if (msglevel >= 9) {
        printf("\nData collected from %d inputs -- %d events\n",
                numinputs, numevents);
        for (i=0; i<numevents; i++)
            printf("   input %d  time %s  value %s\n", event[i].gate,
                    Time2Str(event[i].time),
                    Val2Str(event[i].val));
    }

    assert(gate->function >= input  &&  gate->function <= xor);
    stub = stubs[gate->function];
```

```
    i = 0;
    numout = 0;
    while (i < numevents) {
        /* Loop invarients:                                            */
        /*             i ==> next "input" event to handle              */
        /*             numout ==> place for the next "output" event    */
        /*             event[i] is at a diffent time than event[i-1]   */
        prevtime = event[i].time;
        while (i < numevents  &&  prevtime == event[i].time) {
            inputs[ event[i].gate-1 ] = event[i].val;
            i++;
        }
        Oevents[numout].tl = prevtime;
        Oevents[numout].val  = (*stub)(inputs, numinputs);
        numout++;
    }

    seq = CollapseSequence(Oevents, numout);
    seq = AdjustTime(seq, gate->function);
    gate->seq = seq;

    /* Check for any hazard generation                                 */
    if (seq.val >= st0) {
        /* A hazard has been detected -- whether to output it          */
        /* depends on the msglevel and whether the hazard is on        */
        /* an output line.                                             */
        if (gate->function == output  &&  msglevel >= 1) {
            printf("*** Error: %s on output line %s\n",
                    Seq2Str(seq), gate->name);
        }
        else if (msglevel >= 4) {
            printf("*** Error: %s on the output of gate %s\n",
                    Seq2Str(seq), gate->name);
        }
    }

}
```

C.12   READTIME.C

```
/*===================================================================*/
/*                                                                   */
/* READTIME -- read a file of propagation delay information.         */
/*                                                                   */
/* This function is used to read the TIMEINFO file that contains     */
/* propagation delay information.                                    */
/*                                                                   */
/* See the SYMSIM User's Guide for information about the format      */
/* of this file.                                                     */
/*                                                                   */
/*                                                                   */
/* Copyright (c) 1988, 1989                                          */
/*          Neil Erdwien and Kansas State University                 */
/*          All rights reserved.                                     */
/*                                                                   */
```

- 101 -

```
/*===================================================================*/


/*==================================== Standard C include files */
#ifdef ANSI
#include <stddef.h>
#include <stdlib.h>
#endif
#include <stdio.h>
#include <ctype.h>


/*============================== Program-specific include files */
#include "symsim.h"


/*==================================== Global external variables */
TIME    tPLHmin[MAXGTYPES];
TIME    tPLHmax[MAXGTYPES];
TIME    tPHLmin[MAXGTYPES];
TIME    tPHLmax[MAXGTYPES];


/*==================================== Private function prototypes */
#ifdef ANSI
static TIME read1(FILE *fp);
#endif


/*===================================================================*/
/*                                                                   */
/* read1 -- read a single time value                                 */
/*                                                                   */
/* This function reads one time value from the input file.  The      */
/* value is scaled into an integer format and returned to the        */
/* caller.                                                           */
/*                                                                   */
/* If an invalid number is found, zero is returned.                  */
/*                                                                   */
/*===================================================================*/

static
TIME
read1(fp)

FILE *fp;

{
    float tempfloat;
    char  *string;

    string = GetToken(fp);

    /* Assign a default in case the GetToken failed.            */
    tempfloat = 0.0;
    sscanf(string, "%f", &tempfloat);

    return 10.0 * tempfloat + 0.5;
```

- 102 -

```
      }

void
ReadTimeInfo(filename)

char *filename;

{
   FILE  *fp;
   GTYPE function;
   int   c;

   fp = fopen(filename, "r");
   if (fp == NULL)
      fatal("Error opening time information file '%s'", filename);

   for (function = input; function <= output; function++) {
      tPLHmin[function] = 0;
      tPLHmax[function] = 0;
      tPHLmin[function] = 0;
      tPHLmax[function] = 0;
   }

   /* Echo the first line of the file to stdout.  This line      */
   /* contains the "title" of the time information file.         */
   while (1) {
      c = getc(fp);
      if (c == '\n'  ||  c == EOF)
         break;
      if (msglevel >= 4)
         putchar(c);
   }
   if (msglevel >= 4)
      putchar('\n');

   for (function = inverter; function <= xor; function++) {
      tPLHmin[function] = readl(fp);
      tPLHmax[function] = readl(fp);
      tPHLmin[function] = readl(fp);
      tPHLmax[function] = readl(fp);
   }

   fclose(fp);

   if (msglevel >= 7) {
      puts(
"\n   Function    tPLHmin    tPLHmax    tPHLmin    tPHLmax");

      for (function = inverter; function <= xor; function++) {
         printf("   %8s",   Func2Str(function));
         printf("   %8s",   Time2Str(tPLHmin[(int)function]) );
         printf("   %8s",   Time2Str(tPLHmax[(int)function]) );
         printf("   %8s",   Time2Str(tPHLmin[(int)function]) );
         printf("   %8s\n", Time2Str(tPHLmax[(int)function]) );
      }
      puts("");
   }
```

- 103 -

}

C.13 SEQ2STR.C

```
/*====================================================================*/
/*                                                                    */
/* Seq2Str -- convert a SEQUENCE into a character string              */
/*                                                                    */
/* This function takes a single argument, a value of type             */
/* SEQUENCE, and converts it into a printable character string        */
/* meant for human reading.                                           */
/*                                                                    */
/* If an invalid value is sent in, the string "[invalid]" is          */
/* returned.                                                          */
/*                                                                    */
/* Seq2Name returns a pointer to the character string                 */
/* representation.                                                    */
/*                                                                    */
/* Seq2Name returns the address of an internal static buffer and      */
/* hence the value returned must be used befora the next call.        */
/* This causes problems when printing two values in a single          */
/* printf call -- printf calls this function twice and then           */
/* prints the result.  The solution is to use separate printf         */
/* calls.                                                             */
/*                                                                    */
/*                                                                    */
/* Copyright (c) 1988, 1989                                           */
/*            Neil Erdwien and Kansas State University                */
/*            All rights reserved.                                    */
/*                                                                    */
/*====================================================================*/


/*==================================== Standard C include files */
#ifdef ANSI
#include <stddef.h>
#include <stdlib.h>
#endif
#include <string.h>
#include <stdio.h>


/*================================= Program-specific include files */
#include "symsim.h"


char *
Seq2Str(seq)

SEQUENCE saq;

{
    static char buffer[64];
    char  *p;
```

- 104 -

```
    p = strcpy(buffer, Val2Str(seq.val));

    if (seq.val > one  && seq.val != unk) {
        /* There is time information for this sequence.            */
        if (seq.tA)
            p = strcat(p, " tA+[");
        else
            p = strcat(p, " [");

        p = strcat(p, Time2Str(seq.t1));
        p = strcat(p, ",");
        p = strcat(p, Time2Str(seq.t2));
        p = strcat(p, "]");
    }

    return buffer;

}
```

C.14   SNAP.C

```
/*===================================================================*/
/*                                                                   */
/* Snap -- display the contents of the circuit web                   */
/*                                                                   */
/* The Snap function is a debugging tool.  It can be called at        */
/* any time and it displays the status of the circuit being          */
/* analyzed.  The circuit itself and the status of the analysis      */
/* are unchanged by this procedure.                                  */
/*                                                                   */
/*                                                                   */
/* Copyright (c) 1988, 1989                                          */
/*          Neil Erdwien and Kansas State University                 */
/*          All rights reserved.                                     */
/*                                                                   */
/*===================================================================*/


/*=================================== Standard C include files */
#ifdef ANSI
#include <stddef.h>
#include <stdlib.h>
#endif
#include <stdio.h>
#include <assert.h>


/*================================ Program-specific include files */
#include "symsim.h"


/*================================ Private function prototypes */
#ifdef ANSI
static void SnapRecursive(GBLOK *gate, int indent);
#endif
```

- 105 -

```
/*====================================================================*/
/*                                                                    */
/* SnapRecursive -- perform a recursive depth-first traversal         */
/*                                                                    */
/* This recursive function actually performs the traversal.           */
/* Information about each gate visited is listed.  Because the         */
/* circuit web may contain loops, the VISITED field of each gate       */
/* and the VisitedValue global variable are used to prevent            */
/* repetition.                                                        */
/*                                                                    */
/*====================================================================*/

static
void
SnapRecursive(gate, indent)

GBLOK *gate;
int indent;

{
    int    i;
    GLUE   *outptr;

    assert(gate != NULL);

    indent = indent % 40;

    for (i=0; i<indent; i++)        /* Indent to the indicated level  */
        putchar(' ');

    printf("function = %s, name = %s, causes = %x\n",
            Func2Str(gate->function), gate->name, gate->causes[0]);

    for (i=0; i<indent; i++)        /* Indent to the indicated level  */
        putchar(' ');

    if (gate->visited != VisitedValue) {
        /* This node hasn't been visited yet...                       */
        gate->visited = VisitedValue;
        printf("Output (%s) is connected to:\n",
                Seq2Str(gate->seq));
        for (outptr = gate->olist;
             outptr != NULL;
             outptr = outptr->next) {
            SnapRecursive(outptr->gate, indent + 3);
        }
    } else {
        printf("...previously listed...\n");
    }
}


void
Snap(Iptr)

GBLOK *Iptr[];
```

```
{
    int    i;

    VisitedValue++;
    puts("");

    for (i = 0; Iptr[i] != NULL; i++)
        SnapRecursive(Iptr[i], 0);

    puts("");

}
```

C.15   STUBS.C

```
/*===================================================================*/
/*                                                                   */
/* STUBS -- stubs to perform simple logic functions                  */
/*                                                                   */
/* This file contains a series of functions that implement the       */
/* basic logic functions used by SYMSIM.  These stubs are only       */
/* needed for truth table creation.                                  */
/*                                                                   */
/* These stubs are used on operands from the 5-level sequence        */
/* system, hence they do not have to evaluate the effect of          */
/* hazards on the inputs.                                            */
/*                                                                   */
/*                                                                   */
/* Copyright (c) 1988, 1989                                          */
/*           Neil Erdwien and Kansas State University                */
/*           All rights reserved.                                    */
/*                                                                   */
/*===================================================================*/


/*======================================= Standard C include files */
#ifdef ANSI
#include <stddef.h>
#include <stdlib.h>
#endif
#include <stdio.h>
#include <math.h>
#include <assert.h>
#include <malloc.h>
#include <string.h>


/*================================ Program-specific include files */
#include "symsim.h"


/*================================ Global external variables */
VALUE    (*stubs[])
#ifdef ANSI
```

- 107 -

```
                        (VALUE inputs[], int numinputs)
    #else
                        ()
    #endif
            = {
            stubINP,       /* input  */
            stubINP,       /* output */
            stubINV,
            stubAND,
            stubOR,
            stubNAND,
            stubNOR,
            stubXOR};

VALUE    invert[] = {
         one,
         zero,
         down,
         up,
         unk,
         st1,
         st0,
         dy1,
         dy0
};


/*-------------------------------------------------------------------*/
/*                                                                   */
/* stubINP -- simulate an input (or an output) line                  */
/*                                                                   */
/* This simulation simply returns the input value.                   */
/*                                                                   */
/*-------------------------------------------------------------------*/
VALUE
stubINP(inputs, numinputs)

VALUE inputs[];
int numinputs;

{

    if (numinputs != 1)
        fatal("A multiple-input input has been found");

    assert(inputs[0] >= zero  &&  inputs[0] <= unk);

    /* Always just return the input value                   */
    return inputs[0];

}


/*-------------------------------------------------------------------*/
/*                                                                   */
/* stubINV -- simulate a single input inverter                       */
/*                                                                   */
```

```
/*------------------------------------------------------------------*/
VALUE
stubINV(inputs, numinputs)

VALUE inputs[];
int numinputs;

{

    if (numinputs != 1)
       fatal("A multiple-input inverter has been found");

    assert(inputs[0] >= zero  &&  inputs[0] <= unk);

    /* Use a table lookup to invert the logic level            */
    return invert[inputs[0]];

}


/*------------------------------------------------------------------*/
/*                                                                  */
/* stubAND -- simulate a single multiple input AND gate             */
/*                                                                  */
/*------------------------------------------------------------------*/
VALUE
stubAND(inputs, numinputs)

VALUE inputs[];
int numinputs;

{
    int   i;
    int   rising, falling;

    for (i=0; i<numinputs; i++)
       assert(inputs[i] >= zero  &&  inputs[i] <= unk);

    /* If any of the inputs is 0, the output is 0              */
    for (i=0; i<numinputs; i++)
       if (inputs[i] == zero)
           return zero;

    /* If any of the inputs is unk, the output is unk          */
    for (i=0; i<numinputs; i++)
       if (inputs[i] == unk)
           return unk;

    /* If both a rising and a falling edge are found,          */
    /* a dynamic 0 hazard exists                               */
    rising  = 0;
    falling = 0;
    for (i=0; i<numinputs; i++) {
       if (inputs[i] == up) rising  = 1;
       if (inputs[i] == down)  falling = 1;
    }
```

- 109 -

```
   if (rising == 1  &&  falling == 1)
      return unk;

   /* If one or more of a single type of edge is found,          */
   /* the output matches                                         */
   if (rising == 1)
      return up;

   if (falling == 1)
      return down;

   /* If all else falls through, all the inputs must have been 1 */
   return one;

}


/*------------------------------------------------------------------*/
/*                                                                  */
/* stubOR -- simulate a single multiple input OR gate              */
/*                                                                  */
/*------------------------------------------------------------------*/

VALUE
stubOR(inputs, numinputs)

VALUE inputs[];
int numinputs;

{
   int   i;
   int   rising, falling;

   for (i=0; i<numinputs; i++)
      assert(inputs[i] >= zero  &&  inputs[i] <= unk);

   /* If any of the inputs is 1, the output is 1                  */
   for (i=0; i<numinputs; i++)
      if (inputs[i] == one)
         return one;

   /* If any of the inputs is unk, the output is unk             */
   for (i=0; i<numinputs; i++)
      if (inputs[i] == unk)
         return unk;

   /* If both a rising and a falling edge are found, a static 1  */
   /* hazard exists                                              */
   rising  = 0;
   falling = 0;
   for (i=0; i<numinputs; i++) {
      if (inputs[i] == up) rising  = 1;
      if (inputs[i] == down)  falling = 1;
   }

   if (rising == 1  &&  falling == 1)
      return unk;
```

```
   /* If one or more of a single type of edge is found,          */
   /* the output matches                                          */
   if (rising == 1)
      return up;

   if (falling == 1)
      return down;

   /* If all else falls through, all the inputs must have been 0  */
   return zero;

}


/*-------------------------------------------------------------------*/
/*                                                                   */
/* stubNAND -- simulate a single multiple input NAND gate            */
/*                                                                   */
/*-------------------------------------------------------------------*/

VALUE
stubNAND(inputs, numinputs)

VALUE inputs[];
int numinputs;

{
   int   i;
   int   rising, falling;

   for (i=0; i<numinputs; i++)
      assert(inputs[i] >= zero  &&  inputs[i] <= unk);

   /* If any of the inputs is 0, the output is 1                  */
   for (i=0; i<numinputs; i++)
      if (inputs[i] == zero)
         return one;

   /* If any of the inputs is unk, the output is unk              */
   for (i=0; i<numinputs; i++)
      if (inputs[i] == unk)
         return unk;

   /* If both a rising and a falling edge are found, a dynamic 1  */
   /* hazard exists                                               */
   rising  = 0;
   falling = 0;
   for (i=0; i<numinputs; i++) {
      if (inputs[i] == up) rising  = 1;
      if (inputs[i] == down)  falling = 1;
   }

   if (rising == 1  &&  falling == 1)
      return unk;

   /* If one or more of a single type of edge is found, the       */
   /* output is inverted                                          */
   if (rising == 1)
```

- 111 -

```
      return down;

   if (falling == 1)
      return up;

   /* If all else falls through, all the inputs must have been 1  */
   return zero;

}


/*--------------------------------------------------------------------*/
/*                                                                  */
/* stubNOR -- simulate a single multiple input NOR gate            */
/*                                                                  */
/*--------------------------------------------------------------------*/

VALUE
stubNOR(inputs, numinputs)

VALUE inputs[];
int numinputs;

{
   int    i;
   int    rising, falling;

   for (i=0; i<numinputs; i++)
      assert(inputs[i] >= zero  &&  inputs[i] <= unk);

   /* If any of the inputs is 1, the output is 0                   */
   for (i=0; i<numinputs; i++)
      if (inputs[i] == one)
         return zero;

   /* If any of the inputs is unk, the output is unk              */
   for (i=0; i<numinputs; i++)
      if (inputs[i] == unk)
         return unk;

   /* If both a rising and a falling edge are found, a static 0   */
   /* hazard exists                                               */
   rising  = 0;
   falling = 0;
   for (i=0; i<numinputs; i++) {
      if (inputs[i] == up) rising  = 1;
      if (inputs[i] == down)  falling = 1;
   }

   if (rising == 1  &&  falling == 1)
      return unk;

   /* If one or more of a single type of edge is found, the       */
   /* output is inverted                                          */
   if (rising == 1)
      return down;

   if (falling == 1)
```

- 112 -

```
      return up;

   /* If all else falls through, all the inputs must have been 1  */
   return zero;

}


/*-------------------------------------------------------------------*/
/*                                                                   */
/* stubXOR -- simulate a single multiple input XOR gate              */
/*                                                                   */
/*-------------------------------------------------------------------*/

VALUE
stubXOR(inputs, numinputs)

VALUE inputs[];
int numinputs;

{
   int    i;
   int    rising, falling;
   int    parity;

   for (i=0; i<numinputs; i++)
      assert(inputs[i] >= zero  &&  inputs[i] <= unk);

   /* If any of the inputs is unk, the output is unk                 */
   for (i=0; i<numinputs; i++)
      if (inputs[i] == unk)
         return unk;

   /* If both a rising and a falling edge are found, a static 0      */
   /* hazard exists                                                  */
   rising  = 0;
   falling = 0;
   parity  = 0;
   for (i=0; i<numinputs; i++) {
      if (inputs[i] == one)   parity  = (++parity) & 1;
      if (inputs[i] == up)    rising++;
      if (inputs[i] == down)  falling++;
   }

   if (rising  &&  falling)
      return unk;

   if (rising == 1)
      if (parity)
         return down;
      else
         return up;

   if (falling == 1)
      if (parity)
         return up;
      else
         return down;
```

- 113 -

```
   /* If all else falls through, all the inputs must have been    */
   /* either 0 or 1.  In this case, the parity count is the right */
   /* value.                                                      */
   return parity;

}
```

C.16   TABLE.C

```
/*===================================================================*/
/*                                                                   */
/* TABLE -- simple symbol table handling functions                   */
/*                                                                   */
/* This file contains several functions that together manage         */
/* symbol tables.  This is used within SYMSIM to keep track of       */
/* the names given to the nodes in the Trivial Circuit Format.       */
/*                                                                   */
/* Currently, simple sequential search is used to search the         */
/* table.  If tables of more than 100 nodes are expected, this       */
/* should be changed to use a hash table.  Such changes could        */
/* be done within this file and not be visible to the caller.        */
/*                                                                   */
/* To use these functions, the table.h include file must be          */
/* included in the calling function.                                 */
/*                                                                   */
/* The TableCreate function is used to create a table.  If           */
/* successful, a pointer to the table is returned.  The caller       */
/* should treat this pointer like a file handle, i.e., it is         */
/* passed to other table functions but is not meaningful             */
/* otherwise.                                                        */
/*                                                                   */
/* The TableAdd function adds a new entry to the table.              */
/* Associated with each entry is a variable length key and a         */
/* user-supplied pointer.  When the table entry is subsequently      */
/* used, the caller supplies the key and the table retrieves the     */
/* pointer.                                                          */
/*                                                                   */
/* The TableLookup function searches the table for an entry with     */
/* the given key.  If found, the pointer stored when the entry       */
/* was added is returned.                                            */
/*                                                                   */
/* Finally, the TableFree function can be used to release the        */
/* memory associated with a table.                                   */
/*                                                                   */
/*                                                                   */
/* Copyright (c) 1988, 1989                                          */
/*            Neil Erdwien and Kansas State University               */
/*            All rights reserved.                                   */
/*                                                                   */
/*===================================================================*/


/*==================================== Standard C include files */
#ifdef ANSI
```

- 114 -

```
#include <stddef.h>
#include <stdlib.h>
#endif
#include <stdio.h>
#include <malloc.h>
#include <string.h>


/*================================= Program-specific include files */
#include "symsim.h"
#include "table.h"


TABLE *
TableCreate()

{

    TABLE *tptr;

    tptr = (TABLE *) malloc(sizeof(TABLE));
    if (tptr == NULL)
        fatal("Error allocating a TABLE");
    tptr->anchor = NULL;

    return tptr;
}


int
TableAdd(tptr, key, infoptr)

TABLE *tptr;
char *key;
void *infoptr;

{

    TABLELINK *p;

    p = (TABLELINK *) malloc(sizeof(TABLELINK));
    if (p == NULL)
        fatal("Error allocating a TABLELINK");
    p->next = tptr->anchor;
    tptr->anchor = p;

    p->key = strdup(key);
    p->data = infoptr;

    return 0;
}


void
*TableLookup(tptr, key)

TABLE *tptr;
char *key;
```

- 115 -

```
{
    TABLELINK *p;

    for (p = tptr->anchor; p != NULL; p = p->next) {
        if (stricmp(p->key, key) == 0)
            return p->data;
    }

    return NULL;

}


void
TableFree(tptr)

TABLE *tptr;

{
    TABLELINK *p, *p2;

    for (p = tptr->anchor; p != NULL; ) {
        free(p->key);
        p2 = p->next;
        free(p);
        p = p2;
    }

    free(tptr);

}
```

C.17   TABLE.H

```
/*=====================================================================*/
/*                                                                     */
/* TABLE -- simple symbol table handling functions                     */
/*                                                                     */
/* This header file declares several functions that are used to        */
/* handle a symbol table.                                              */
/*                                                                     */
/* See the implementation in TABLE.C for details.                      */
/*                                                                     */
/*                                                                     */
/* Copyright (c) 1988, 1989                                            */
/*             Neil Erdwien and Kansas State University                */
/*             All rights reserved.                                    */
/*                                                                     */
/*=====================================================================*/


/*==================================== Data Structure Declarations */
typedef struct tablelink {
    struct tablelink *next;
    char  *key;
```

- 116 -

```
    void  *data;
} TABLELINK;

typedef struct {
    TABLELINK *anchor;
} TABLE;


/*================================== External function declarations */

#ifdef ANSI

TABLE    *TableCreate(void);
int       TableAdd(TABLE *tptr, char *key, void *infoptr);
void     *TableLookup(TABLE *tptr, char *key);
void      TableFree(TABLE *tptr);

#else

TABLE    *TableCreate();
int       TableAdd();
void     *TableLookup();
void      TableFree();

#endif
```

C.18   TIME2STR.C

```
/*====================================================================*/
/*                                                                    */
/* Time2Str -- convert a time value to a string                       */
/*                                                                    */
/* This function formats an internal time variable into a             */
/* printable form.  Currently, time values are stored as integers     */
/* in units of tenths of the units used in the TIMEINFO file --       */
/* typically nanoseconds.  This function divides by  10 to            */
/* compensate.                                                        */
/*                                                                    */
/* Any units may be used.                                             */
/*                                                                    */
/* Time2Str is passed a single argument -- the time value to be       */
/* formatted -- and returns a pointer to the character string         */
/* representation.                                                    */
/*                                                                    */
/* Time2Str returns the address of an internal static buffer and      */
/* hence the value returned must be used before the next call.        */
/* This causes problems when printing two values in a single          */
/* printf call -- printf calls this function twice and then           */
/* prints the result.  The solution is to use separate printf         */
/* calls.                                                             */
/*                                                                    */
/*                                                                    */
/* Copyright (c) 1988, 1989                                           */
/*            Neil Erdwien and Kansas State University                */
/*            All rights reserved.                                    */
```

- 117 -

```
/*                                                                   */
/*==================================================================*/


/*=================================== Standerd C include files */
#ifdef ANSI
#include <stddef.h>
#include <stdlib.h>
#endif
#include <stdio.h>


/*============================== Program-specific include files */
#include "symsim.h"


char *
Time2Str(t)

TIME t;

{
    static cher buffer[32];

    if (t == INITIAL)   /* If it is the special "initial" value... */
        return "I";     /* ...return "I"                           */

    if (t >= MAXTIME)   /* If it is the special "maximum" value... */
        return "MAXTIME";/* ...return "MAXTIME".                   */

    sprintf(buffer, "%.lf", t/10.0);    /* Format the time value   */

    return buffer;
}



C.19    TOKEN.C

/*==================================================================*/
/*                                                                   */
/* GetToken -- read the next token from a file                       */
/*                                                                   */
/* This function takes a single argument, a file pointer of type     */
/* (FILE *), and reads the next token from the file.  The file is    */
/* assumed to be open for input.                                     */
/*                                                                   */
/* A token is defined as a series of non-whitespace characters,      */
/* delimited by whitespace.  Whitespace is defined by the C          */
/* library's isspace function, i.e., 0x09-0x0d or 0x20 on ASCII      */
/* machines.                                                         */
/*                                                                   */
/* Tokens Ionger than MAXTOKENSIZE are truncated with no error       */
/* indication.  The string returned is ALWAYS terminated by a        */
/* '\0', even when truncation occurs.                                */
/*                                                                   */
/* The file being read cen contain comments which are skipped        */
```

```
/* by GetToken.  Comments start with a COMMENTCHAR, currently   */
/* a '#', and continue to the end of the line.  Comments are    */
/* syntactically a blank.                                       */
/*                                                              */
/* At EOF, a null string is returned.                           */
/*                                                              */
/*                                                              */
/* Copyright (c) 1988, 1989                                     */
/*          Neil Erdwien and Kansas State University            */
/*          All rights reserved.                                */
/*                                                              */
/*============================================================*/


/*================================= Standard C include files */
#ifdef ANSI
#include <stddef.h>
#include <stdlib.h>
#endif
#include <stdio.h>
#include <ctype.h>


/*============================= Program-specific include files */
#include "symsim.h"


/*============================= Manifest constant definitions */
#define MAXTOKENSIZE 128
#define COMMENTCHAR '#'


/*============================= Private function prototypes */
#ifdef ANSI
static void SkipComments(FILE *fp);
#endif


/*============================================================*/
/*                                                              */
/* SkipComments -- skip comments and white space                */
/*                                                              */
/* This private function finds the beginning of the next token by */
/* skipping any number of blanks, other wite space characters,   */
/* and comments.                                                */
/*                                                              */
/* On exit, the next character to be read will be the first non- */
/* blank character of the next token -- or the file will be at   */
/* EOF.                                                         */
/*                                                              */
/*============================================================*/


static
void
SkipComments(fp)

FILE  *fp;
```

```
{
    int   c;

    while (1) {

        c = getc(fp);

        if (isspace(c))
            /* Skip the character */;

        else if (c == COMMENTCHAR)
            /* We've started a comment -- skip to tha end of tha line */
            while (1) {
                c = getc(fp);
                if (c == '\n'  ||  c == EOF)
                    break;
            }

        else
            /* Found an interesting character -- exit                    */
            /* Note that EOF is treated like any other character.        */
            break;

    }

    ungetc(c, fp);

}


char *
GetToken(fp)

FILE  *fp;

{
    int   c;
    static char buffer[MAXTOKENSIZE+1];
    char  *bufptr;

    SkipComments(fp);

    bufptr = buffer;
    while (1) {
        c = getc(fp);
        if (feof(fp)  ||  isspace(c)  ||  c == COMMENTCHAR)
            break;
        if (bufptr < &buffer[MAXTOKENSIZE])
            *(bufptr++) = c;
    }

    /* There will always be room in the buffer for an ending '\0'   */
    /* because of the '+1' in the daclaration of the buffer.        */
    *bufptr = '\0';

    return buffer;

}
```

C.20   TSIM1.C

```
/*===================================================================*/
/*                                                                   */
/* Sim1 -- simulate a single logic element                           */
/*                                                                   */
/* This function takes a single argument, a pointer to the GBLOK     */
/* representing the gate to be simulated.  The input values to       */
/* the gate are found by traversing the circuit web.                 */
/*                                                                   */
/* There are two files that both contain functions named Sim1.       */
/* This file, TSIM1, contains the table-driven version that          */
/* combines inputs in pairs to calculata the result.                 */
/* LONGSIM1 contains a version that goes through the raw 5-level      */
/* sequence calculation to obtain the result.  LONGSIM1 is used      */
/* by TTABLE to calculate the tables used by TSIM1.                   */
/*                                                                   */
/* At one time, LONGSIM1 and TSIM1 were interchangeable by simply    */
/* linking one or the other.  TSIM1 has since been enhanced and      */
/* must be used by SYMSIM.  Conversely, TTABLE must use LONGSIM1.    */
/*                                                                   */
/*                                                                   */
/* Copyright (c) 1988, 1989                                           */
/*           Neil Erdwien and Kansas State University                 */
/*           All rights reserved.                                     */
/*                                                                   */
/*===================================================================*/


/*==================================== Standard C include files */
#ifdef ANSI
#include <stddef.h>
#include <stdlib.h>
#endif
#include <stdio.h>
#include <math.h>
#include <assert.h>
#include <malloc.h>
#include <string.h>


/*==================================== Program-specific include files */
#include "symsim.h"


/*==================================== Global external variables */
extern   TIME   tPLHmin[MAXGTYPES];
extern   TIME   tPLHmax[MAXGTYPES];
extern   TIME   tPHLmin[MAXGTYPES];
extern   TIME   tPHLmax[MAXGTYPES];


/*==================================== Private static variables */
typedef struct {
   VALUE result;
   short LeadOp1;
   short TrailOp1;
} S1MENTRY;
```

- 121 -

```c
     typedef SIMENTRY SIMTABLE[9][9][6];

     /* These are the truth tables calculated by TTABLE.            */
     static
     SIMTABLE TblAND = {
     #include "and.tbl"
     };

     static
     SIMTABLE TblOR = {
     #include "or.tbl"
     };

     static
     SIMTABLE TblXOR = {
     #include "xor.tbl"
     };

     static
     VALUE    invert[] = {
              one,
              zero,
              down,
              up,
              unk,
              stl,
              st0,
              dyl,
              dy0
     };


     /*================================== Private function prototypes */
     #ifdef ANSI
     static SEQUENCE AdjustTime(SEQUENCE seq, GTYPE function);
     #endif


     /*===================================================================*/
     /*                                                                   */
     /* AdjustTime -- adjust time values for the delay of this element    */
     /*                                                                   */
     /* The AdjustTime function alters the time values of a sequence      */
     /* to include the propagation delay of the element being             */
     /* simulated.                                                        */
     /*                                                                   */
     /*===================================================================*/

     static
     SEQUENCE
     AdjustTime(seq, function)

     SEQUENCE seq;
     GTYPE function;

     {

         switch (seq.val) {
```

```
         case   zero:
         case   one:
         case   unk:
            break;
         case   up:
         case   dy0:
            seq.tl = seq.tl + tPLHmin[function];
            seq.t2 = seq.t2 + tPLHmax[function];
            break;
         case   down:
         case   dyl:
            seq.tl = seq.tl + tPHLmin[function];
            seq.t2 = seq.t2 + tPHLmax[function];
            break;
         case   st0:
            seq.tl = seq.tl + tPLHmin[function];
            seq.t2 = seq.t2 + tPHLmax[function];
            break;
         case   stl:
            seq.tl = seq.tl + tPHLmin[function];
            seq.t2 = seq.t2 + tPLHmax[function];
            break;
         default:
            fatal("Unknown logic value in AdjustTime");
    }

    return seq;

}


void
Siml(gate)

GBLOK *gate;

{

    GLUE      *pi;
    SEQUENCE  result;
    SEQUENCE  operand1;
    SEQUENCE  operand2;
    int       InvertResult;
    int       alignment;
    SIMTABLE  *TablePtr;
    SIMENTRY  *TableEntry;

    InvertResult = 0;

    switch (gate->function) {

        case input:
            /* There isn't much work in simulating an input!      */
            return;

        case output:
            /* An output is simulated only to trigger the error   */
            /* message at the end.                                */
```

```
        pi       = gate->ilist;
        operand1 = pi->gate->seq;
        result   = operand1;
        break;

    case inverter:
        /* An inverter is necessarily a single-input gate,    */
        /* hence handled separately here.                     */
        pi       = gate->ilist;
        operand1 = pi->gate->seq;
        rasult   = operand1;
        InvertResult = 1;
        break;

    case nand:
        InvertResult = 1;
    case and:
        /* Microsoft C issues a warning message on the following */
        /* assignment statment -- it is OK.                     */
        TablePtr = &TblAND[0][0][0];
        break;

    case nor:
        InvertResult = 1;
    case or:
        /* Microsoft C issues a warning messaga on the following */
        /* assignment statment -- it is OK.                     */
        TablePtr = &TblOR[0][0][0];
        break;

    case xor:
        /* Microsoft C issues a warning message on tha following */
        /* assignment statment -- it is OK.                     */
        TablePtr = &TblXOR[0][0][0];
        break;

}


if (gata->function > inverter) {
    /* This section handles the multiple-input gates, i.e.,  */
    /* AND, OR, NAND, NOR, and XOR gates.                    */
    /* These are simulated two inputs at a time.  Initially, */
    /* the loop is primed by pretending that the first input */
    /* value is the result of a previous iteration.  The     */
    /* following loop executes once for a 2 input gate, twice */
    /* for a 3 input gate, atc.                              */
    pi = gate->ilist;
    result = pi->gate->seq;
    pi = pi->next;

    while (pi != NULL) {

        /* Loop invariant: RESULT contains tha result so far of */
        /* the simulation.   PI points at the next GLUE block   */
        /* for the next input.                                  */
        operand1 = result;
        oparand2 = pi->gate->seq;
```

```
        pi = pi->next;

        /* Brute-force code to turn the time values into a 0-5    */
        /* alignment value.                                       */
        if (operand2.tl < operandl.tl) {

            /* Must be cases 0, 1, or 5.                          */
            if (operand2.t2 <= operandl.tl)
                alignment = 0;
            else if (operand2.t2 <= operandl.t2)
                alignment = 1;
            else
                alignment = 5;

        } else {

            /* Must be cases 2, 3, or 4.                          */
            if (operand2.t2 <= operandl.t2)
                alignment = 2;
            else if (operand2.tl < operandl.t2)
                alignment = 3;
            else
                alignment = 4;

        }


        TableEntry = &((*TablePtr)[operandl.val]
                                  [operand2.val][elignment]);

        result.val = TableEntry->result;
        if (TableEntry->LeadOpl)
            result.tl = operandl.tl;
        else
            result.tl = operand2.tl;

        if (TableEntry->TrailOpl)
            result.t2 = operendl.t2;
        else
            result.t2 = operend2.t2;

    }
}


assert(result.val >= zero  &&  result.val <= dyl);

if (InvertResult)
    result.val = invert[result.val];

result = AdjustTime(result, gate->function);

gete->seq = result;

/* Check for any hazard generation                              */
if (result.vel >= st0) {
    /* A hazard has been detected -- whether to output it       */
    /* depends on the msglevel end whether the hezard is on en  */
```

```
        /* output line.                                               */
        if (gate->function == output  &&  msglevel >= 1) {
            printf("*** Error: %s on output line %s\n",
                    Seq2Str(result), gate->name);
            puts("    Caused by:");
            AssmPrt();
        }
        else if (msglevel >= 4) {
            printf("*** Error: %s on the output of gate %s\n",
                    Seq2Str(result), gate->name);
            puts("    Caused by:");
            AssmPrt();
        }
    }

}
```

C.21   TTABLE.C

```
/*===================================================================*/
/*                                                                   */
/* TTABLE -- build a truth table based on 9-value logic              */
/*                                                                   */
/* This program performs a detailed simulation for a two-input       */
/* logic element.  Each 9-level input value is separated into a      */
/* series of up to three 5-level values.  The resulting sequence     */
/* (including time values) is simulated.  The resulting output       */
/* sequence is turned into a 9-level value.                          */
/*                                                                   */
/* The output of this program is a table in a form suitable for      */
/* inclusion in a C program.  In a sense, this program is just       */
/* a bootstrap.  Once this program creates the needed tables,        */
/* further simuletion uses the tables rather than the long method    */
/* used here.                                                        */
/*                                                                   */
/* The tables produced by this program match those given by          */
/* Fantauzzi in "An Algebraic Model for the Analysis of Logical      */
/* Circuits", IEEE Transactions on Computers, Vol. C-23, No. 6,      */
/* June 1974, pp. 576-581.                                           */
/*                                                                   */
/* This program is not a general-purpose simulator; it exists        */
/* only to produce tables for the general-purpose simulator.         */
/* As such, the circuit input to this program is restricted to       */
/* circuits with exactly two inputs and one output and no            */
/* internal memory.                                                  */
/*                                                                   */
/*                                                                   */
/* Copyright (c) 1988, 1989                                          */
/*          Neil Erdwien and Kansas State University                 */
/*          All rights reserved.                                     */
/*                                                                   */
/*===================================================================*/


/*=================================== Standard C include files */
```

- 126 -

```
#ifdef ANSI
#include <stddef.h>
#include <stdlib.h>
#endif
#include <stdio.h>
#include <math.h>
#include <assert.h>
#include <malloc.h>
#include <string.h>


/*================================== Program-specific include files */
#include "symsim.h"


/*================================== Global external variables */
int       msglevel;
GBLOK     *Iptr[MAXINPUTS];
GBLOK     *Optr[MAXINPUTS];
char      *TimeInfoFilename;
char      *CircuitFilename;
int       Acount   = 0;
int       Amax;
int       VisitedValue = 0;


/*================================== External function declarations */
#ifdef ANSI
extern int  getopt(int argc, char **argv, char *options);
#endif


/*================================== Private function prototypes */
#ifdef ANSI
static void ProcessOpts(int argc, char **argv);
static void DoI(SEQUENCE v1, SEQUENCE v2);
#endif


/*=================================================================*/
/*                                                                 */
/* ProcessOpts -- handle the options passed to the main program    */
/*                                                                 */
/* The getopt function is used to parse the option string.         */
/* Global variables are used to pass the results back.             */
/* Default values for every option are set here.                   */
/*                                                                 */
/*=================================================================*/

static
void
ProcessOpts(argc, argv)

int argc;
char **argv;

{
```

- 127 -

```
    int         c;
    extern char *optarg;
    extern int   optind, opterr;
    int          errflg = 0;

    /* Set default values for all options.                              */
    TimeInfoFilename = "timeinfo.0";
    CircuitFilename = "and.tcf";
    msglevel = 0;

    while ((c = getopt(argc, argv, "m:t:")) != -1)
        switch (c) {
        case 'm':
            msglevel = atoi(optarg);
            break;
        case 't':
            TimeInfoFilename = optarg;
            break;
        case '?':
            errflg++;
        }

    if (errflg) {
        fprintf(stderr,
                "Usage: TTABLE [-m msglevel] [-t timefile] file\n");
        exit(2);
    }

    if (optind < argc)
        CircuitFilename = argv[optind];

    if (optind+1 < argc)
        printf("Extra parameters ignored\n");

}


/*===================================================================*/
/*                                                                   */
/* Dol -- simulate and single input combination                      */
/*                                                                   */
/* The Dol subroutine simulates the circuit for a single setting     */
/* of the input variables.  The results of the simulation are        */
/* written to  stdout .                                              */
/*                                                                   */
/*===================================================================*/
static
void
Dol(v1, v2)

SEQUENCE v1;
SEQUENCE v2;

{

    SEQUENCE result;
    GBLOK    *g2;
```

- 128 -

```
GBLOK    *gate;

Iptr[0]->seq = vl;
Iptr[1]->seq = v2;

Forward(Iptr[0]);

Forward(Iptr[1]);

g2 = Optr[0];    /* Point G2 at the first output              */
result = g2->ilist->gate->seq;
printf("%5s,", Val2Str(result.val));

if (result.val == zero  ||  result.val == one  ||
    result.val == unk)
    printf("0,0,");

else {

    if (result.t1 == vl.t1)
        printf("1,");
    else if (result.t1 == v2.t1)
        printf("0,");
    else {
        printf("\nError in simulation -- time = %s\n",
                Time2Str(result.t1));
        printf("\n%d, %d\n", vl.t1, v2.t1);
        exit(1);
    }

    if (result.t2 == vl.t2)
        printf("1");
    else if (result.t2 == v2.t2)
        printf("0");
    else {
        printf("\nError in simulation -- time = %s\n",
                Time2Str(result.t2));
        exit(1);
    }

    if (vl.val != dyl  ||
        v2.val != dyl  ||
        v2.t2-v2.t1 < vl.t2-vl.t1)
        printf(",");
}

/* Pop the stack until the previous assumption is found. */
/* Each gate stacked is set back to 'unknown'.           */
while(1) {
    gate = pop;
    if (gate == NULL)
        break;
    gate->seq.val = unk;
    gate->seq.tA  = 0;
}

}
```

- 129 -

```
void
main(argc, argv)

int argc;
char **argv;

{
    int      rc;
    SEQUENCE v1, v2;

    ProcessOpts(argc, argv);

    ReadTimeInfo(TimeInfoFilename);

    rc = GetCircuit(CircuitFilename, Iptr, Optr);
    if (rc != 0)
        fatal("Error reading circuit '%s'", CircuitFilename);

    if (lptr[0] == NULL || lptr[1] == NULL || lptr[2] != NULL)
        fatal("Circuit must have exactly two inputs");

    if (msglevel >= 9)
        Snap(lptr);

    v1.tA = 0;
    v2.tA = 0;

    printf("/*              ");
    printf(
"    1            2            3            4            5            6    */\n");

    printf("/*              ");
    printf(
"   XXX          XXX          XXX          XXX          XXX          XXX  */\n");

    printf("/*  X     Y     ");
    printf(
"YY           YY           Y            YY           YY   YYYYY */\n");

    /* Loop over all possible combinations of the input variables. */
    for (v1.val = zero; v1.val <= dy1; v1.val++) {
        for (v2.val = zero; v2.val <= dy1; v2.val++) {
            printf("/*%4s,%4s*/", Val2Str(v1.val), Val2Str(v2.val));

            /* The VALUES of the input signals are set, but their    */
            /* relative timing is not.  There are six possible       */
            /* cases -- each is tried in turn.  The actual values    */
            /* used here must match those tested inside Do1.         */

            v1.t1 = 100;
            v1.t2 = 200;
            v2.t1 =  50;
            v2.t2 =  70;
            Do1(v1, v2);

            v1.t1 = 100;
```

- 130 -

```
            v1.t2 = 200;
            v2.t1 =  90;
            v2.t2 = 110;
            Do1(v1, v2);

            v1.t1 = 100;
            v1.t2 = 200;
            v2.t1 = 110;
            v2.t2 = 130;
            Do1(v1, v2);

            v1.t1 = 100;
            v1.t2 = 200;
            v2.t1 = 190;
            v2.t2 = 210;
            Do1(v1, v2);

            v1.t1 = 100;
            v1.t2 = 200;
            v2.t1 = 210;
            v2.t2 = 230;
            Do1(v1, v2);

            v1.t1 = 100;
            v1.t2 = 200;
            v2.t1 =  50;
            v2.t2 = 250;
            Do1(v1, v2);

            printf("\n");
        }
    }

    exit(0);
}
```

C.22   VAL2STR.C

```
/*===================================================================*/
/*                                                                   */
/* Val2Str -- convert a VALUE into a cheracter string neme           */
/*                                                                   */
/* This function takes a single argument, a value of type VALUE,     */
/* and converts it into a printable character string meant for       */
/* human reading.                                                    */
/*                                                                   */
/* 1f an invalid value is sent in, the string "[invalid]" is         */
/* returned.                                                         */
/*                                                                   */
/* Val2Neme returns a pointer to the character string                */
/* representation.                                                   */
/*                                                                   */
/* Val2Name returns the address of an internal static buffer and     */
/* hence the value returned must be used before the next call.       */
/* This ceuses problems when printing two values in a single         */
```

```
/* printf call -- printf calls this function twice and then       */
/* prints the result.  The solution is to usa separate printf     */
/* calls.                                                         */
/*                                                                */
/*                                                                */
/* Copyright (c) 1988, 1989                                       */
/*          Neil Erdwien and Kansas State University              */
/*          All rights reserved.                                  */
/*                                                                */
/*===============================================================*/


/*===================================== Standard C includa files */
#ifdef ANS1
#include <stddef.h>
#include <stdlib.h>
#endif
#include <stdio.h>


/*================================ Program-specific include files */
#include "symsim.h"


/*================================ Manifest constant definitions */
#define MAXENTRIES 10


/*===================================== Private static variables */
static char *names[MAXENTRIES] = {
    "zero",
    "one",
    "up",
    "down",
    "unk",
    "st0",
    "stl",
    "dy0",
    "dyl",
    "[invalid]"
};


char *
Val2Str(value)

VALUE value;

{
    int i;

    i = (int) value;    /* Convert the valua to a subscript...   */
                        /* ...ensure that it is in ranga...      */
    if (i < 0  ||  i >= MAXENTRIES)
        i = MAXENTRIES-1;

    return names[i];    /* ...and return the corresponding entry. */
}
```

C.23 SYMSIM Makefile

```
# MS-DOS makefile for SYMSIM

# This makefile is used with the Microsoft MAKE command.
# SYMSIM will be build by simply entering 'MAKE SYMSIM'.

.c.obj:
        CL /c /AS /DANS1 $*.c

symsim.obj:     symsim.c

getcirc.obj:    getcirc.c

token.obj:      token.c

table.obj:      table.c

getopt.obj:     getopt.c

snap.obj:       snap.c

func2str.obj:   func2str.c

val2str.obj:    val2str.c

seq2str.obj:    seq2str.c

time2str.obj:   time2str.c

fatal.obj:      fatal.c

tsim1.obj:      tsim1.c

forward.obj:    forward.c

assume.obj:     assume.c

readtime.obj:   readtime.c

addinput.obj:   addinput.c

AssmPrt.obj:    AssmPrt.c

symsim.exe: symsim.obj symsim.lnk \
        getcirc.obj getopt.obj snap.obj func2str.obj val2str.obj \
        seq2str.obj time2str.obj forward.obj fatal.obj tsim1.obj \
        readtime.obj assume.obj token.obj table.obj addinput.obj \
        AssmPrt.obj
        LINK @symsim.lnk
```

C.24   TTABLE Makefile

```
# MS-DOS makefile for TTABLE

# This makefile is used with the Microsoft MAKE command.
# TTABLE will be build by simply entering 'MAKE TTABLE'.

.c.obj:
        CL /c /AS /OANSI $*.c

ttable.obj:      ttable.c

getopt.obj:      getopt.c

getcirc.obj:     getcirc.c

token.obj:       token.c

snap.obj:        snap.c

table.obj:       table.c

val2str.obj:     val2str.c

func2str.obj:    func2str.c

seq2str.obj:     seq2str.c

time2str.obj:    time2str.c

stubs.obj:       stubs.c

fatal.obj:       fatal.c

longsiml.obj:    longsiml.c

forward.obj:     forward.c

readtime.obj:    readtime.c

ttable.exe: ttable.obj ttable.lnk \
        getopt.obj table.obj token.obj getcirc.obj val2str.obj \
        func2str.obj seq2str.obj time2str.obj stubs.obj fatal.obj \
        longsiml.obj forward.obj readtime.obj snap.obj
        LINK @ttable.lnk
```

C.25   Unix Makefile

```
# This makefile is used for Un*x systems.
# The CFLAGS macro should contain any options to the compiler needed
# for the various systems.
# The symbol BSD must be defined for systems derived from Berkeley
# Unix.  SYSV must be defined for System V systems.

# This makefile has been tested on SUNOS 4.0 on a Sun 3/60,
# and AT&T System V on an AT&T 3B15.

#Uncomment the following CFLAGS definition for the Sun.
CFLAGS = -O -DBSD

#Uncomment the following CFLAGS definition for the 3B15.
#CFLAGS = -DSYSV

all:    ttable symsim

ttable: ttable.o table.o token.o \
        getcirc.o func2str.o val2str.o seq2str.o time2str.o \
        stubs.o fatal.o longsiml.o forward.o readtime.o snap.o
        cc $(CFLAGS) -o $@ ttable.o table.o token.o \
        getcirc.o func2str.o val2str.o seq2str.o time2str.o \
        stubs.o fatal.o longsiml.o forward.o readtime.o snap.o

symsim: and.tbl or.tbl xor.tbl \
        symsim.o getcirc.o snap.o func2str.o val2str.o seq2str.o \
        time2str.o forward.o fatal.o tsiml.o readtime.o assume.o \
        bitops.o token.o table.o addinput.o assmprt.o
        cc $(CFLAGS) -o $@ \
        symsim.o getcirc.o snap.o func2str.o val2str.o seq2str.o \
        time2str.o forward.o fatal.o tsiml.o readtime.o assume.o \
        bitops.o token.o table.o addinput.o assmprt.o

and.tbl: ttable
        ttable -m0 and.tcf >and.tbl

or.tbl: ttable
        ttable -m0 or.tcf >or.tbl

xor.tbl: ttable
        ttable -m0 xor.tcf >xor.tbl
```

Appendix D

Sample Trivial Circuit Format Files

D.1 HAZARD1.TCF

```
# This circuit is the simple static 0-hazard generator.

element I1 input
    INV1 AND1
element INV1 inverter
    AND1
element AND1 and
    O1
element O1 output
```

D.2 TWOWAY.TCF

```
# This circuit is the simple static 0-hazard generator
# with two inverters in the top path.  This demonstrates
# a hazard for both transitions.

element I1 input
    INV1 INV2
element INV1 inverter
    AND1
element INV2 inverter
    INV3
element INV3 inverter
    AND1
element AND1 and
    O1
element O1 output
```

D.3   HAZBLOCK.TCF

```
# This circuit is the standard static 0-hazard generator with
# its output "blocked" by an AND gate.

element I1 input
    INV1 AND1
element AND1 and
    INV2
element INV1 inverter
    AND1
element INV2 inverter
    AND2
element I2 input
    AND2
element AND2 and
    D1
element D1 output
```

D.4   AND.TCF

```
# A single AND gate
# This circuit is used by TTABLE to generate tables used by SYMSIM.

element D1 output
element GATE and
    O1
element I1 input
    GATE
element I2 input
    GATE
```

D.5   OR.TCF

```
# A single OR gate
# This circuit is used by TTABLE to generate tables used by SYMSIM.

element D1 output
element GATE OR
    D1
element I1 input
    GATE
element I2 input
    GATE
```

D.6   XOR.TCF

```
# A single XOR gate
# This circuit is used by TTABLE to generate tables used by SYMSIM.

element O1 output
element GATE XOR
    O1
element I1 input
    GATE
element I2 input
    GATE
```

Appendix E

Sample Time Information Files

E.1   TIMEINFO.ALS

74ALS TTL prop. delays, Logic Databook, vol. 11, Nat. Semi. 1984

The first line of this file is a title line which is printed
when the file is read.

Characters (like these) following a pound sign (#) are ignored.

Six sets of data follow, one for each type of gate used by
SYMSIM. The data must be listed in the order below, i.e.,
Inverter, AND, OR, NAND, NOR, and finally XOR.

# TPLHmin	TPLHmax	TPHLmin	TPHLmax	Function	IC Number
3.0	11.0	2.0	8.0	# Inverter	DM74ALS04
4.0	14.0	3.0	10.0	# AND	DM74ALS08
3.0	14.0	3.0	12.0	# OR	DM74ALS32
3.0	11.0	2.0	8.0	# NAND	DM74ALS00
3.0	12.0	3.0	10.0	# NOR	DM74ALS02

This next one is worst-case for the two cases of whether
the other input is high or low.

| 3.0 | 17.0 | 3.0 | 12.0 | # XOR | DM74ALS32 |

The above numbers are for VCC=4.5V-5.5V, RL=500 ohms, CL=50 pF

E.2   TIMEINFO.AS

74AS TTL prop. delays, Logic Databook, vol. 11, Nat. Semi. 1984

The first line of this file is a title line which is printed
when the file is read.

Characters (like these) following a pound sign (#) are ignored.

Six sets of data follow, one for each type of gate used by
SYMSIM. The data must be listed in the order below, i.e.,
Inverter, AND, OR, NAND, NOR, and finally XOR.

# TPLHmin	TPLHmax	TPHLmin	TPHLmax	Function	IC Number
1.0	5.0	1.0	4.0	# Inverter	DM74AS04

- 139 -

```
      1.0      5.5      1.0      5.5      # ANO          OM74AS08
      1.0      5.8      1.0      5.8      # OR           DM74AS32
      1.0      4.5      1.0      4.0      # NANO         DM74AS00
      1.0      4.5      1.0      4.5      # NOR          OM74AS02
# This next one is a totel guess -- it is missing in the book!
      1.0      5.8      1.0      5.8      # XOR          DM74AS32

# The above numbers are for VCC=4.5V-5.5V, RL=500 ohms, CL=50 pF
```

E.3   TIMEINFO.LS

74LS TTL prop. delays, Logic Databook, vol. Il, Nat. Semi. 1984

```
# The first line of this file is a title line which is printed
# when the file is read.

# Characters (like these) following a pound sign (#) are ignored.

# Six sets of data follow, one for each type of gate used by
# SYMS1M.  The data must be listed in the order below, i.e.,
# Inverter, ANO, OR, NANO, NOR, and finally XOR.

# TPLHmin  TPLHmax  TPHLmin  TPHLmax      Function    IC Number
      4.0     15.0      4.0     15.0      # Inverter   DM74LS04
      6.0     18.0      5.0     18.0      # ANO        DM74LS08
      4.0     15.0      4.0     15.0      # OR         OM74LS32
      4.0     15.0      4.0     15.0      # NANO       OM74LS00
      4.0     18.0      4.0     15.0      # NOR        OM74LS02
# This next one is worst-case for tha two cases of
# whether the other input is high or low.
# Furthermore, the mins are guesses -- they aren't in the book.
      4.0     23.0      4.0     21.0      # XOR        OM74LS32

# The above numbers are for VCC = 5.0V, RL = 2K ohms, CL = 50 pF
```

E.4   TIMEINFO.0

Zero propagation delay model

```
# The first line of this file is a title line which is printed
# when the file is read.

# Characters (like these) following a pound sign (#) are ignored.

# Six sets of data follow, one for each type of gate used by
# SYMS1M.  The data must be listed in the order below, i.e.,
# Inverter, ANO, OR, NANO, NOR, and finally XOR.

# This is mythical zero-delay information.

# TPLHmin  TPLHmax  TPHLmin  TPHLmax      Function    IC Number
      0.0      0.0      0.0      0.0      # Inverter
      0.0      0.0      0.0      0.0      # AND
```

```
0.0     0.0     0.0     0.0     # OR
0.0     0.0     0.0     0.0     # NAND
0.0     0.0     0.0     0.0     # NOR
0.0     0.0     0.0     0.0     # XOR
```

BIBLIOGRAPHY

1. E. B. Eichelberger, "Hazard detection in combinational and sequential switching circuits", *IBM Journal*, Vol. 9, No. 3, March 1965, pp. 90-99.

2. M. Yoeli and S. Rinon, "Application of ternary algebra to the study of static hazards", *Journal of the Association of Computing Machinery*, Vol. 11, January 1964, pp. 84-87.

3. D. E. Muller, "Treatment of transition signals in electronic switching circuits by algebraic methods", *IRE Trans. Electron. Comput.*, Vol. EC-8, September 1959, p. 401.

4. M. A. Breuer and A. D. Friedman, *Diagnosis and Reliable Design of Digital Systems*, Rockville, Maryland: Computer Science Press, 1976.

5. G. Fantauzzi, "An algebraic model for the analysis of logical circuits", *IEEE Transactions on Computers*, Vol. C-23, No. 6, June 1974, pp. 576-581.

6. E. J. McCluskey, *Logic Design Principles*, Englewood Cliffs, New Jersey: Prentice-Hall, 1986.

7. N. P. Jouppi, "Timing analysis and performance improvement of MOS VLSI designs", *IEEE Transactions of Computer-Aided Design*, Vol. CAD-6, No. 4, July 1987, pp. 650-665.

8. National Semiconductor Corporation, *Logic Databook, Volume II*, Santa Clara, CA: 1984.

9.   B. Harding, "Timing verifiers branch out to overcome analysis constraints", *Computer Design*, May 1, 1989, pp. 53-56.

10.   R. W. Hon, "Dynamic analysis tools", *Computer Aids for VLSI Design*, Addison-Wesley Publishing Company, 1987.

11.   Huffman D. A., "The synthesis of sequential circuits", *J. Franklin Inst.*, Vol. 257, 1954, pp. 161-191 and 275-303.

12.   Texas Instruments Incorporated, *The TTL Data Book for Design Engineers*, Dallas, TX: 1981.

Automatic Detection of Hazards

in

Digital Logic Circuits

by

Neil C. Erdwien
BSEE, Kansas State University, 1984

AN ABSTRACT OF A THESIS

submitted in partial fulfillment of the
the requirements for the degree

MASTER OF SCIENCE

Electrical Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1989

ABSTRACT

This thesis describes SYMSIM, a symbolic simulator for digital logic circuits that is capable of detecting all static and dynamic hazards in combinational circuits. SYMSIM is based on a conventional simulator with a realistic propagation delay model that provides distinct propagation delays for different gate types as well as separate rise and fall times. An exhaustive search algorithm drives the simulator to examine all hazard possibilities. Time values are maintained as symbolic values during simulation, hence the name SYMSIM. Hazards are propagated through the network, allowing hazard detections to be suppressed if the hazard never reaches an output. These methods could be extended to sequential circuits although the execution time would suffer.